User's Guide

SIESTA trunk-625–linres-623

July 24, 2017

http://www.uam.es/siesta

Contributors to SIESTA

Emilio Artacho	*CIC-Nanogune and University of Cambridge*
José María Cela	*Barcelona Supercomputing Center*
Julian D. Gale	*Curtin University of Technology, Perth*
Alberto García	*Institut de Ciència de Materials, CSIC, Barcelona*
Javier Junquera	*Universidad de Cantabria, Santander*
Richard M. Martin	*University of Illinois at Urbana-Champaign*
Pablo Ordejón	*Centre de Investigació en Nanociència i Nanotecnologia, (CSIC-ICN), Barcelona*
Daniel Sánchez-Portal	*Unidad de Física de Materiales, Centro Mixto CSIC-UPV/EHU, San Sebastián*
José M. Soler	*Universidad Autónoma de Madrid*

The SIESTA project was initiated by Pablo Ordejon (then at the Univ. de Oviedo), and Jose M. Soler and Emilio Artacho (Univ. Autonoma de Madrid, UAM). The development team was then joined by Alberto Garcia (then at Univ. del Pais Vasco, Bilbao), Daniel Sanchez-Portal (UAM), and Javier Junquera (Univ. de Oviedo and later UAM), and sometime later by Julian Gale (then at Imperial College, London). In 2007 Jose M. Cela (Barcelona Supercomputing Center, BSC) became a core developer.

The current TranSIESTA module within SIESTA is developed by Nick R. Papior (then at Technical University of Denmark) and Mads Brandbyge. The original TranSIESTA module was developed by Pablo Ordejon and Jose L. Mozos (then at ICMAB-CSIC), and Mads Brandbyge, Kurt Stokbro, and Jeremy Taylor (Technical Univ. of Denmark).

Other contributors (we apologize for any omissions):

Eduardo Anglada, Thomas Archer, Luis C. Balbas, Xavier Blase, Ramon Cuadrado, Michele Ceriotti, Raul de la Cruz, Gabriel Fabricius, Marivi Fernandez-Serra, Jaime Ferrer, Chu-Chun Fu, Sandra Garcia, Victor M. Garcia-Suarez, Georg Huhs, Rogeli Grima, Rainer Hoft, Sergio Illera, Jorge Kohanoff, Richard Korytar, In-Ho Lee, Lin Lin, Nicolas Lorente, Miquel Llunell, Eduardo Machado, Maider Machado, Jose Luis Martins, Volodymyr Maslyuk, Juana Moreno, Frederico Dutilh Novaes, Micael Oliveira, Nick Rübner Papior, Magnus Paulsson, Oscar Paz, Andrei Postnikov, Tristana Sondon, Rafi Ullah, Andrew Walker, Andrew Walkingshaw, Toby White, Francois Willaime, Chao Yang.

O.F. Sankey, D.J. Niklewski and D.A. Drabold made the FIREBALL code available to P. Ordejon. Although we no longer use the routines in that code, it was essential in the initial development of

SIESTA, which still uses many of the algorithms developed by them.

Contents

1 INTRODUCTION

This Reference Manual contains descriptions of all the input, output and execution features of SIESTA*, but is not really a tutorial introduction to the program. Interested users can find tutorial material prepared for* SIESTA *schools and workshops at the project's web page* `http://www.uam.es/siesta`.

NOTE: See the description of changes in the logic of the SCF loop

SIESTA (Spanish Initiative for Electronic Simulations with Thousands of Atoms) is both a method and its computer program implementation, to perform electronic structure calculations and *ab initio* molecular dynamics simulations of molecules and solids. Its main characteristics are:

- It uses the standard Kohn-Sham selfconsistent density functional method in the local density (LDA-LSD) and generalized gradient (GGA) approximations, as well as in a non local functional that includes van der Waals interactions (VDW-DF).
- It uses norm-conserving pseudopotentials in their fully nonlocal (Kleinman-Bylander) form.
- It uses atomic orbitals as a basis set, allowing unlimited multiple-zeta and angular momenta, polarization and off-site orbitals. The radial shape of every orbital is numerical and any shape can be used and provided by the user, with the only condition that it has to be of finite support, i.e., it has to be strictly zero beyond a user-provided distance from the corresponding nucleus. Finite-support basis sets are the key for calculating the Hamiltonian and overlap matrices in $O(N)$ operations.
- Projects the electron wavefunctions and density onto a real-space grid in order to calculate the Hartree and exchange-correlation potentials and their matrix elements.
- Besides the standard Rayleigh-Ritz eigenstate method, it allows the use of localized linear combinations of the occupied orbitals (valence-bond or Wannier-like functions), making the computer time and memory scale linearly with the number of atoms. Simulations with several hundred atoms are feasible with modest workstations.
- It is written in Fortran 95 and memory is allocated dynamically.
- It may be compiled for serial or parallel execution (under MPI).

It routinely provides:

- Total and partial energies.
- Atomic forces.
- Stress tensor.
- Electric dipole moment.
- Atomic, orbital and bond populations (Mulliken).
- Electron density.

And also (though not all options are compatible):

- Geometry relaxation, fixed or variable cell.
- Constant-temperature molecular dynamics (Nose thermostat).
- Variable cell dynamics (Parrinello-Rahman).
- Spin polarized calculations (collinear or not).
- k-sampling of the Brillouin zone.
- Local and orbital-projected density of states.
- COOP and COHP curves for chemical bonding analysis.
- Dielectric polarization.
- Vibrations (phonons).
- Band structure.
- Ballistic electron transport under non-equilibrium (through TranSIESTA)

Starting from version 3.0, SIESTA includes the TranSIESTA module. TranSIESTA provides the ability to model open-boundary systems where ballistic electron transport is taking place. Using TranSIESTA one can compute electronic transport properties, such as the zero bias conductance and the I-V characteristic, of a nanoscale system in contact with two electrodes at different electrochemical potentials. The method is based on using non equilibrium Greens functions (NEGF), that are constructed using the density functional theory Hamiltonian obtained from a given electron density. A new density is computed using the NEGF formalism, which closes the DFT-NEGF self consistent cycle.

For more details on the formalism, see the main TranSIESTA reference cited below. A section has been added to this User's Guide, that describes the necessary steps involved in doing transport calculations, together with the currently implemented input options.

References:

- "Unconstrained minimization approach for electronic computations that scales linearly with system size" P. Ordejón, D. A. Drabold, M. P. Grumbach and R. M. Martin, Phys. Rev. B **48**, 14646 (1993); "Linear system-size methods for electronic-structure calculations" Phys. Rev. B **51** 1456 (1995), and references therein.

  Description of the order-N eigensolvers implemented in this code.

- "Self-consistent order-N density-functional calculations for very large systems" P. Ordejón, E. Artacho and J. M. Soler, Phys. Rev. B **53**, 10441, (1996).

  Description of a previous version of this methodology.

- "Density functional method for very large systems with LCAO basis sets" D. Sánchez-Portal, P. Ordejón, E. Artacho and J. M. Soler, Int. J. Quantum Chem., **65**, 453 (1997).

  Description of the present method and code.

- "Linear-scaling ab-initio calculations for large and complex systems" E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

  Description of the numerical atomic orbitals (NAOs) most commonly used in the code, and brief review of applications as of March 1999.

- "Numerical atomic orbitals for linear-scaling calculations" J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B **64**, 235111, (2001).

  Improved, soft-confined NAOs.

- "The SIESTA method for ab initio order-N materials simulation" J. M. Soler, E. Artacho, J.D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal, J. Phys.: Condens. Matter **14**, 2745-2779 (2002)

  Extensive description of the SIESTA method.

- "Computing the properties of materials from first principles with SIESTA", D. Sánchez-Portal, P. Ordejón, and E. Canadell, Structure and Bonding **113**, 103-170 (2004).

  Extensive review of applications as of summer 2003.

- "Density-functional method for nonequilibrium electron transport", Mads Brandbyge, Jose-Luis Mozos, Pablo Ordejón, Jeremy Taylor, and Kurt Stokbro, Phys. Rev. B **65**, 165401 (2002).

  Description of the TranSIESTA method.

For more information you can visit the web page `http://www.uam.es/siesta`.

2 COMPILATION

2.1 The build directory

Rather than using the top-level Src directory as building directory, the user has to use an ad-hoc building directory (by default the top-level `Obj` directory, but it can be any (new) directory in the top level). The building directory will hold the object files, module files, and libraries resulting from the compilation of the sources in `Src`. The `VPATH` mechanism of modern `make` programs is used. This scheme has many advantages. Among them:

- The Src directory is kept pristine.
- Many different object directories can be used concurrently to compile the program with different compilers or optimization levels.

If you just want to compile the program, go to `Obj` and issue the command:

```
sh ../Src/obj_setup.sh
```

to populate this directory with the minimal scaffolding of makefiles, and then make sure that you create or generate an appropriate `arch.make` file (see below, in Sec. 2.2). Then, type

```
make
```

The executable should work for any job. (This is not exactly true, since some of the parameters in the atomic routines are still hardwired (see `Src/atmparams.f`), but those would seldom need to be changed.)

To compile utility programs (those living in `Util`), you can just simply use the provided makefiles, typing "make" as appropriate.

2.1.1 Multiple-target compilation

The mechanism described here can be repeated in other directories at the same level as Obj, with different names. In this way one can compile as many different versions of the SIESTA executable as needed (for example, with different levels of optimization, serial, parallel, debug, etc), by working in separate building directories.

Simply provide the appropriate arch.make, and issue the setup command above. To compile utility programs, you need to use the form:

```
make OBJDIR=ObjName
```

where `ObjName` is the name of the object directory of your choice. Be sure to type `make clean` before attempting to re-compile a utility program.

(The pristine Src directory should be kept "clean", without objects, or else the compilation in the build directories will get confused)

2.2 The arch.make file

The compilation of the program is done using a `Makefile` that is provided with the code. This `Makefile` will generate the executable for any of several architectures, with a minimum of tuning required from the user and encapsulated in a separate file called `arch.make`.

You are strongly encouraged to look at `Obj/DOCUMENTED-TEMPLATE.make` for information about the fine points of the `arch.make` file. There are two sample make files for compilation of SIESTA with `gfortran` and `ifort` named `Obj/gfortran.make` and `Obj/intel.make`, respectively. Please use those as guidelines for creating the final `arch.make`.

2.3 Parallel

To achieve a parallel build of SIESTA one should first determine which type of parallelism one requires. It is advised to use MPI for calculations with moderate number of cores. If one requires eXa-scale parallelism SIESTA provides hybrid parallelism using both MPI and OpenMP.

2.3.1 MPI

MPI is a message-passing interface which enables communication between equivalently executed binaries. This library will thus duplicate all non-distributed data such as local variables etc.

To enable MPI in SIESTA the compilation options are required to be changed accordingly, here is the most basic changes to the `arch.make` for standard binary names

```
CC = mpicc
FC = mpifort # or mpif90
MPI_INTERFACE = libmpi_f90.a
MPI_INCLUDE = .
FPPFLAGS += -DMPI
```

Subsequently one may run SIESTA using the `mpirun`/`mpiexec` commands:

```
mpirun -np <> siesta RUN.fdf
```

where `<>` is the number of cores used.

2.3.2 OpenMP

OpenMP is shared memory parallelism. It typically does not infer any memory overhead and may be used if memory is scarce and the regular MPI compilation is crashing due to insufficient memory.

To enable OpenMP, simply add this to your `arch.make`

```
# For GNU compiler
FFLAGS += -fopenmp
LIBS += -fopenmp
# or, for Intel compiler < 16
FFLAGS += -openmp
LIBS += -openmp
# or, for Intel compiler >= 16
FFLAGS += -qopenmp
LIBS += -qopenmp
```

The above will yield the most basic parallelism using OpenMP. However, the BLAS/LAPACK libraries which is the most time-consuming part of SIESTA also requires to be threaded, please see Sec. 2.4 for correct linking.

Subsequently one may run SIESTA using OpenMP through the environment variable `OMP_NUM_THREADS` which determine the number of threads/cores used in the execution.

```
OMP_NUM_THREADS=<> siesta RUN.fdf
# or (bash)
export OMP_NUM_THREADS=<>
siesta RUN.fdf
# or (csh)
setenv OMP_NUM_THREADS <>
siesta RUN.fdf
```

where <> is the number of threads/cores used.

If SIESTA is also compiled using MPI it is more difficult to obtain a good performance. Please refer to your local cluster how to correctly call MPI with hybrid parallelism. An example for running SIESTA with good performance using OpenMPI > 1.8.2 *and* OpenMP on a machine with 2 sockets and 8 cores per socket, one may do:

```
# MPI = 2 cores, OpenMP = 8 threads per core (total=16)
mpirun --map-by ppr:1:socket:pe=8 \
   -x OMP_NUM_THREADS=8 \
   -x OMP_PROC_BIND=true siesta RUN.fdf

# MPI = 4 cores, OpenMP = 4 threads per core (total=16)
mpirun --map-by ppr:2:socket:pe=4 \
   -x OMP_NUM_THREADS=4 \
   -x OMP_PROC_BIND=true siesta RUN.fdf

# MPI = 8 cores, OpenMP = 2 threads per core (total=16)
mpirun --map-by ppr:4:socket:pe=2 \
   -x OMP_NUM_THREADS=2 \
   -x OMP_PROC_BIND=true siesta RUN.fdf
```

If using only 1 thread per MPI core it is advised to compile SIESTA without OpenMP. As such it may be advantageous to compile SIESTA in 3 variants; OpenMP-only (small systems), MPI-only (medium to large systems) and MPI+OpenMP (large+ systems).

2.4 Library dependencies

SIESTA makes use of several libraries. Here we list a set of libraries and how each of them may be added to the compilation step (`arch.make`).

SIESTA is distributed with scripts that install the most useful libraries. These installation scripts may be located in the `Docs/` folder with names: `install_*.bash`. Currently SIESTA is shipped with these installation scripts:

- `install_netcdf4.bash`; installs NetCDF with full CDF4 support. Thus it installs zlib, hdf5 *and* NetCDF C and Fortran.
- `install_flook.bash`; installs `flook` which enables interaction with Lua and SIESTA.

Note that these scripts are guidance scripts and users are encouraged to check the mailing list for or seek help there in non-standard. The installation scripts finishes by telling *what* to add to the `arch.make` file to correctly link the just installed libraries.

BLAS it is recommended to use a high-performance library (OpenBLAS or MKL library from Intel)

- If you use your *nix distribution package manager to install BLAS you are bound to have a poor performance. Please try and use performance libraries, whenever possible!
- If you do not have the BLAS library you may use the BLAS library shipped with SIESTA. To do so simply add `libsiestaBLAS.a` to the `COMP_LIBS` variable.

To add BLAS to the `arch.make` file you need to add the required linker flags to the `LIBS` variable in the `arch.make` file.

Example variables

```
# OpenBLAS:
LIBS += -L/opt/openblas/lib -lopenblas
# or for MKL
LIBS += -L/opt/intel/.../mkl/lib/intel64 -lmkl_blas95_lp64 ...
```

To use the threaded (OpenMP) libraries, change the above linking to

```
# OpenBLAS:
LIBS += -L/opt/openblas/lib -lopenblasp
# or for MKL
LIBS += -L/opt/intel/.../mkl/lib/intel64 -lmkl_blas95_lp64
  -lmkl_<>_thread ...
```

where `<>` is the compiler used (`intel` or `gnu`).

LAPACK it is recommended to use a high-performance library (OpenBLAS[1] or MKL library from Intel)

If you do not have the LAPACK library you may use the LAPACK library shipped with SIESTA. To do so simply add `libsiestaLAPACK.a` to the `COMP_LIBS` variable.

Example variables

```
# OpenBLAS (OpenBLAS will default to build in LAPACK 3.6)
LIBS += -L/opt/openblas/lib -lopenblas
# or for MKL
LIBS += -L/opt/intel/.../mkl/lib/intel64 -lmkl_lapack95_lp64 ...
```

To use the threaded (OpenMP) libraries, change the above linking to

```
# OpenBLAS (OpenBLAS will default to build in LAPACK 3.6)
LIBS += -L/opt/openblas/lib -lopenblasp
# or for MKL
LIBS += -L/opt/intel/.../mkl/lib/intel64 -lmkl_lapack95_lp64
  -lmkl_<>_thread ...
```

where `<>` is the compiler used (`intel` or `gnu`).

ScaLAPACK *Only required for MPI compilation.*

Here one may be sufficient to rely on the NetLIB[2] version of ScaLAPACK.

Example variables

```
# ScaLAPACK
LIBS += -L/opt/scalapack/lib -lscalapack
# or for MKL
LIBS += -L/opt/intel/.../mkl/lib/intel64 -lmkl_scalapack_lp64
         -lmkl_blacs_<>_lp64 ...
```

[1] OpenBLAS enables the inclusion of the LAPACK routines. This is advised.
[2] ScaLAPACKs performance is mainly governed by BLAS and LAPACK.

where <> refers to the MPI version used, (`intelmpi`, `openmpi`, `sgimpt`).

Additionally SIESTA may be compiled with support for several other libraries

fdict This library is shipped with SIESTA and its linking may be enabled by

```
COMP_LIBS += libfdict.a
```

NetCDF It is adviced to compile NetCDF in CDF4 compliant mode (thus also linking with HDF5) as this enables more advanced IO. If you only link against a CDF3 compliant library you will not get the complete feature set of SIESTA.

3 If the CDF3 compliant library is present one may add this to your `arch.make`:

```
LIBS += -L/opt/netcdf/lib -lnetcdff -lnetcdf
FPPFLAGS += -DCDF
```

4 If the CDF4 compliant library is present the HDF5 libraries are also required at link time:

```
LIBS += -L/opt/netcdf/lib -lnetcdff -lnetcdf \
          -lhdf5_fortran -lhdf5 -lz
```

ncdf This library is shipped with SIESTA and its linking is required to take advantage of the CDF4 library functionalities. To use this library, ensure that you can compile SIESTA with CDF4 support. Then proceed by adding the following to your `arch.make`

```
COMP_LIBS += libncdf.a libfdict.a
FPPFLAGS += -DNCDF -DNCDF_4
```

If the NetCDF library is compiled with parallel support one may take advantage of parallel IO by adding this to the `arch.make`

```
FPPFLAGS += -DNCDF_PARALLEL
```

To easily install `NetCDF` please see the installation file: `Docs/install_netcdf4.bash`.

Metis The Metis library may be used in the Order-N code.

Add these flags to your `arch.make` file to enable Metis

```
LIBS += -L/opt/metis/lib -lmetis
FPPFLAGS += -DSIESTA__METIS
```

MUMPS The MUMPS library may currently be used with TranSIESTA.

Add these flags to your `arch.make` file to enable MUMPS

```
LIBS += -L/opt/mumps/lib -lzmumps -lmumps_common <>
FPPFLAGS += -DSIESTA__MUMPS
```

where <> are any libraries that MUMPS depend on.

PEXSI The PEXSI library may be used with SIESTA for exa-scale calculations, see Sec. 6.13. Currently the interface is implemented (tested) as in PEXSI version 0.8.0, 0.9.0 and 0.9.2. If newer versions retain the same interface they may also be used.

To successfully compile SIESTA with PEXSI support one require the PEXSI fortran interface. When installing PEXSI copy the `f_interface.f90` file to the include directory of PEXSI such that the module may be found[3] when compiling SIESTA.

Add these flags to your `arch.make` file to enable PEXSI

```
INCFLAGS += -I/opt/pexsi/include
LIBS += -L/opt/pexsi/lib -lpexsi_linux <>
FPPFLAGS += -DSIESTA__PEXSI
```

where `<>` are any libraries that PEXSI depend on. If one experiences linker failures, one possible solution that may help is

```
LIBS += -lmpi_cxx -lstdc++
```

which is due to PEXSI being a C++ library, and the Fortran compiler is the linker. The exact library name for your MPI vendor may vary.

Additionally the PEXSI linker step may have duplicate objects which can be circumvented by prefixing the PEXSI libraries with

```
LIBS += -Wl,--allow-multiple-definition -lpexsi_linux <>
```

flook SIESTA allows external control via the LUA scripting language. Using this library one may do advanced MD simulations and much more *without* changing any code in SIESTA.

Add these flags to your `arch.make` file to enable `flook`

```
LIBS += -L/opt/flook/lib -lflookall -ldl
COMP_LIBS += libfdict.a
FPPFLAGS += -DSIESTA__FLOOK
```

See `Tests/h2o_lua` for an example on the LUA interface.

To easily install `flook` please see the installation file: `Docs/install_flook.bash`.

3 EXECUTION OF THE PROGRAM

A fast way to test your installation of SIESTA and get a feeling for the workings of the program is implemented in directory `Tests`. In it you can find several subdirectories with pre-packaged FDF files and pseudopotential references. Everything is automated: after compiling SIESTA you can just go into any subdirectory and type `make`. The program does its work in subdirectory `work`, and there you can find all the resulting files. For convenience, the output file is copied to the parent directory. A collection of reference output files can be found in `Tests/Reference`. Please note that small numerical and formatting differences are to be expected, depending on the compiler. (For non-standard execution environments, including queuing systems, have a look at the Scripts in Tests/Scripts, and see also Sec. 2.3.)

Other examples are provided in the `Examples` directory. This directory contains basically `.fdf` files and the appropriate pseudopotential generation input files. Since at some point you will have to generate your own pseudopotentials and run your own jobs, we describe here the whole process by

[3] Optionally the file may be copied to the `Obj` directory where the compilation takes place.

means of the simple example of the water-molecule. It is advisable to create independent directories for each job, so that everything is clean and neat, and out of the SIESTA directory, so that one can easily update version by replacing the whole SIESTA tree. Go to your favorite working directory and:

```
$ mkdir h2o
$ cd h2o
$ cp path-to-package/Examples/H2O/h2o.fdf
```

You need to make the siesta executable visible in your path. You can do it in many ways, but a simple one is

```
$ ln -s path-to-package/Obj/siesta
```

We need to generate the required pseudopotentials. (We are going to streamline this process for this time, but you must realize that this is a tricky business that you must master before using SIESTA responsibly. Every pseudopotential must be thoroughly checked before use. Please refer to the ATOM program manual for details regarding what follows.)

NOTE: The ATOM program is no longer bundled with SIESTA, but academic users can dowload it from the SIESTA webpage at `www.icmab.es/siesta`.

`$ cd path/to/atom/package/`

(Compile the program following the instructions)

`$ cd Tutorial/PS_Generation/O`

`$ cat O.tm2.inp`

This is the input file, for the oxygen pseudopotential, that we have prepared for you. It is in a standard (but ancient and obscure) format that you will need to understand in the future:

```
------------------------------------------------------------
   pg      Oxygen
        tm2  2.0
 n=0  c=ca
       0.0       0.0       0.0       0.0       0.0       0.0
    1    4
    2    0     2.00      0.00
    2    1     4.00      0.00
    3    2     0.00      0.00
    4    3     0.00      0.00
   1.15     1.15     1.15     1.15
------------------------------------------------------------
```

To generate the pseudopotential do the following;

`$ sh ../../Utils/pg.sh O.tm2.inp`

Now there should be a new subdirectory called O.tm2 (O for oxygen) and `O.tm2.vps` (binary) and `O.tm2.psf` (ASCII) files.

```
$ cp O.tm2.psf path-to-working-dir/h2o/O.psf
```

copies the generated pseudopotential file to your working directory. (The unformatted and ASCII files are functionally equivalent, but the latter is more transportable and easier to look at, if you so desire.) The same could be repeated for the pseudopotential for H, but you may as well copy `H.psf` from `Examples/Vps/` to your `h2o` working directory.

Now you are ready to run the program:

```
./siesta < h2o.fdf | tee h2o.out
```

(If you are running the parallel version you should use some other invocation, such as `mpirun -np 2 siesta ...`, but we cannot go into that here — see Sec. 2.3).

After a successful run of the program, you should have several files in your directory including the following:

- fdf.log (contains all the data used, explicit or chosen by default)
- O.ion and H.ion (complete information about the basis and KB projectors)
- h2o.XV (contains positions and velocities)
- h2o.STRUCT_OUT (contains the final cell vectors and positions in "crystallographic" format)
- h2o.DM (contains the density matrix to allow a restart)
- h2o.ANI (contains the coordinates of every MD step, in this case only one)
- h2o.FA (contains the forces on the atoms)
- h2o.EIG (contains the eigenvalues of the Kohn-Sham Hamiltonian)
- h2o.xml (XML marked-up output)

The prefix h2o of all these files is the **SystemLabel** specified in the input h2o.fdf file (see FDF section below). The standard output of the program, that you have already seen passing on the screen, was copied to file h2o.out by the tee command. Have a look at it and refer to the output-explanation section if necessary. You may also want to look at the fdf.log file to see all the default values that siesta has chosen for you, before studying the input-explanation section and start changing them.

Now look at the other data files in `Examples` (all with an .fdf suffix) choose one and repeat the process for it.

3.1 Specific execution options

SIESTA may be executed in different forms. The basic execution form is

```
siesta < RUN.fdf > RUN.out
```

which uses a *pipe* statement. Since 4.1 SIESTA does not require one to pipe in the input file and the input file may instead be specified on the command line.

```
siesta RUN.fdf > RUN.out
```

This allows for SIESTA to accept special flags described in what follows. Each flag may be quoted if it contains spaces, or one may substitute spaces by **:**.

-h print a help instruction and quit

-L Override, temporarily, the **SystemLabel** flag.
`siesta -L Hello`.

-out|-o Specify the output file (instead of printing to the terminal).
`siesta -out RUN.out`.

Additionally TranSIESTA accepts these flags:

-V specify the bias for the current TranSIESTA run.
`transiesta -V 0.25:eV` or `transiesta -V "0.25 eV"` which sets the applied bias to 0.25 eV.

4 THE FLEXIBLE DATA FORMAT (FDF)

The main input file, which is read as the standard input (unit 5), contains all the physical data of the system and the parameters of the simulation to be performed. This file is written in a special format called FDF, developed by Alberto García and José M. Soler. This format allows data to be given in any order, or to be omitted in favor of default values. Refer to documentation in ∼/siesta/Src/fdf for details. Here we offer a glimpse of it through the following rules:

- The FDF syntax is a 'data label' followed by its value. Values that are not specified in the datafile are assigned a default value.
- FDF labels are case insensitive, and characters - _ . in a data label are ignored. Thus, LatticeConstant and lattice_constant represent the same label.
- All text following the # character is taken as comment.
- Logical values can be specified as T, true, .true., yes, F, false, .false., no. Blank is also equivalent to true.
- Character strings should **not** be in apostrophes.
- Real values which represent a physical magnitude must be followed by its units. Look at function fdf_convfac in file ∼/siesta/Src/fdf/fdf.f for the units that are currently supported. It is important to include a decimal point in a real number to distinguish it from an integer, in order to prevent ambiguities when mixing the types on the same input line.
- Complex data structures are called blocks and are placed between '%block label' and a '%end-block label' (without the quotes).
- You may 'include' other FDF files and redirect the search for a particular data label to another file. If a data label appears more than once, its first appearance is used.
- If the same label is specified twice, the first one takes precedence.

- If a label is misspelled it will not be recognized (there is no internal list of "accepted" tags in the program). You can check the actual value used by siesta by looking for the label in the output *fdf.log* file.

These are some examples:

```
SystemName      Water molecule  # This is a comment
SystemLabel     h2o
SpinPolarized        yes
SaveRho
NumberOfAtoms         64
LatticeConstant       5.42 Ang
%block LatticeVectors
         1.000  0.000  0.000
         0.000  1.000  0.000
         0.000  0.000  1.000
%endblock LatticeVectors
KgridCutoff < BZ_sampling.fdf

# Reading the coordinates from a file
%block AtomicCoordinatesAndAtomicSpecies < coordinates.data

# Even reading more FDF information from somewhere else
%include mydefaults.fdf
```

The file *fdf.log* contains all the parameters used by SIESTA in a given run, both those specified in the input fdf file and those taken by default. They are written in fdf format, so that you may reuse them as input directly. Input data blocks are copied to the fdf.log file only if you specify the *dump* option for them.

5 PROGRAM OUTPUT

5.1 Standard output

SIESTA writes a log of its workings to standard output (unit 6), which is usually redirected to an "output file".

A brief description follows. See the example cases in the siesta/Tests directory for illustration.

The program starts writing the version of the code which is used. Then, the input FDF file is dumped into the output file as is (except for empty lines). The program does part of the reading and digesting of the data at the beginning within the `redata` subroutine. It prints some of the information it digests. It is important to note that it is only part of it, some other information being accessed by the different subroutines when they need it during the run (in the spirit of FDF input). A complete list of the input used by the code can be found at the end in the file `fdf.log`, including defaults used by the code in the run.

After that, the program reads the pseudopotentials, factorizes them into Kleinman-Bylander form, and generates (or reads) the atomic basis set to be used in the simulation. These stages are documented in the output file.

The simulation begins after that, the output showing information of the MD (or CG) steps and the SCF cycles within. Basic descriptions of the process and results are presented. The user has the option to customize it, however, by defining different options that control the printing of informations like coordinates, forces, $\vec{k}$ points, etc. The options are discussed in the appropriate sections, but take into account the behavior of the legacy **LongOutput** option, as in the current implementation might silently activate output to the main .out file at the expense of auxiliary files.

LongOutput **false** *(logical)*

SIESTA can write to standard output different data sets depending on the values for output options described below. By default SIESTA will not write most of them. They can be large for large systems (coordinates, eigenvalues, forces, etc.) and, if written to standard output, they accumulate for all the steps of the dynamics. SIESTA writes the information in other files (see Output Files) in addition to the standard output, and these can be cumulative or not.

Setting **LongOutput** to **true** changes the default of some options, obtaining more information in the output (verbose). In particular, it redefines the defaults for the following:

- **WriteKpoints**
- **WriteKbands**
- **WriteCoorStep**
- **WriteForces**
- **WriteEigenvalues**
- **WriteWaveFunctions**
- **WriteMullikenPop**(it sets it to 1)

The specific changing of any of these options has precedence.

5.2 Output to dedicated files

SIESTA can produce a wealth of information in dedicated files, with specific formats, that can be used for further analysis. See the appropriate sections, and the appendix on file formats. Please take into account the behavior of **LongOutput**, as in the current implementation might silently activate output to the main .out file at the expense of auxiliary files.

6 DETAILED DESCRIPTION OF PROGRAM OPTIONS

Here follows a description of the variables that you can define in your SIESTA input file, with their data types and default values. For historical reasons the names of the tags do not have an uniform structure, and can be confusing at times.

Almost all of the tags are optional: SIESTA will assign a default if a given tag is not found when needed (see `fdf.log`).

6.1 General system descriptors

SystemLabel **siesta** *(string)*

A *single* word (max. 20 characters *without blanks*) containing a nickname of the system, used to name output files.

SystemName ⟨**None**⟩ *(string)*

A string of one or several words containing a descriptive name of the system (max. 150 characters).

NumberOfSpecies ⟨**lines in ChemicalSpeciesLabel**⟩ *(integer)*

Number of different atomic species in the simulation. Atoms of the same species, but with a different pseudopotential or basis set are counted as different species.

NOTE: is not required to be set.

NumberOfAtoms ⟨**lines in AtomicCoordinatesAndAtomicSpecies**⟩ *(integer)*

Number of atoms in the simulation.

NOTE: is not required to be set.

%block ChemicalSpeciesLabel ⟨**None**⟩ *(block)*

It specifies the different chemical species that are present, assigning them a number for further identification. SIESTA recognizes the different atoms by the given atomic number.

```
%block ChemicalSpecieslabel
   1   6   C
   2  14   Si
   3  14   Si_surface
%endblock ChemicalSpecieslabel
```

The first number in a line is the species number, it is followed by the atomic number, and then by the desired label. This label will be used to identify corresponding files, namely, pseudopotential file, user basis file, basis output file, and local pseudopotential output file.

This construction allows you to have atoms of the same species but with different basis or pseudopotential, for example.

Negative atomic numbers are used for *ghost* atoms (see **PAO.Basis**).

For atomic numbers over 200 or below −200 you should read **SyntheticAtoms**.

NOTE: this block is mandatory.

%block SyntheticAtoms ⟨**None**⟩ *(block)*

This block is an additional block to complement **ChemicalSpeciesLabel** for special atomic numbers.

Atomic numbers over 200 are used to represent *synthetic atoms* (created for example as a "mixture" of two real ones for a "virtual crystal" (VCA) calculation). In this special case a new **SyntheticAtoms** block must be present to give SIESTA information about the "ground state" of the synthetic atom.

```
%block ChemicalSpeciesLabel
   1   201 ON-0.50000
%endblock ChemicalSpeciesLabel
%block SyntheticAtoms
```

```
   1               # Species index
   2 2 3 4         # n numbers for valence states  with l=0,1,2,3
   2.0 3.5 0.0 0.0 # occupations of valence states with l=0,1,2,3
%endblock SyntheticAtoms
```

Pseudopotentials for synthetic atoms can be created using the `mixps` and `fractional` programs in the `Util/VCA` directory.

Atomic numbers below -200 represent *ghost synthetic atoms*.

%block AtomicMass ⟨**None**⟩ *(block)*

It allows the user to introduce the atomic masses of the different species used in the calculation, useful for the dynamics with isotopes, for example. If a species index is not found within the block, the natural mass for the corresponding atomic number is assumed. If the block is absent all masses are the natural ones. One line per species with the species index (integer) and the desired mass (real). The order is not important. If there is no integer and/or no real numbers within the line, the line is disregarded.

```
%block AtomicMass
   3  21.5
   1  3.2
%endblock AtomicMass
```

The default atomic mass are the natural masses. For *ghost* atoms (i.e. floating orbitals) the mass is 10^{30} a.u.

6.2 Pseudopotentials

SIESTA uses pseudopotentials to represent the electron-ion interaction (as do most plane-wave codes and in contrast to so-called "all-electron" programs). In particular, the pseudopotentials are of the "norm-conserving" kind, and can be generated by the `Atom` program, (see `Pseudo/README.ATOM`). Remember that **all pseudopotentials should be thoroughly tested** before using them. We refer you to the standard literature on pseudopotentials and to the `ATOM` manual for more information. A number of other codes (such as `APE`) can generate pseudopotentials that SIESTA can use directly (typically in the `.psf` format).

The pseudopotentials will be read by SIESTA from different files, one for each defined species (species defined either in block **ChemicalSpeciesLabel**). The name of the files should be:

Chemical_label`.vps` (unformatted) or *Chemical_label*`.psf` (ASCII)

where *Chemical_label* corresponds to the label defined in the **ChemicalSpeciesLabel** block.

6.3 Basis set and KB projectors

6.3.1 Overview of atomic-orbital bases implemented in SIESTA

The main advantage of atomic orbitals is their efficiency (fewer orbitals needed per electron for similar precision) and their main disadvantage is the lack of systematics for optimal convergence, an issue that quantum chemists have been working on for many years. They have also clearly shown that there is no limitation on precision intrinsic to LCAO. This section provides some information about how basis sets can be generated for SIESTA.

It is important to stress at this point that neither the SIESTA method nor the program are bound to the use of any particular kind of atomic orbitals. The user can feed into SIESTA the atomic basis set he/she choses by means of radial tables (see **User.Basis** below), the only limitations being: (*i*) the functions have to be atomic-like (radial functions mutiplied by spherical harmonics), and (*ii*) they have to be of finite support, i.e., each orbital becomes strictly zero beyond some cutoff radius chosen by the user.

Most users, however, do not have their own basis sets. For these users we have devised some schemes to generate basis sets within the program with a minimum input from the user. If nothing is specified in the input file, Siesta generates a default basis set of a reasonable quality that might constitute a good starting point. Of course, depending on the accuracy required in the particular problem, the user has the degree of freedom to tune several parameters that can be important for quality and efficiency. A description of these basis sets and some performance tests can be found in the references quoted below.

"Numerical atomic orbitals for linear-scaling calculations", J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B **64**, 235111, (2001)

An important point here is that the basis set selection is a variational problem and, therefore, minimizing the energy with respect to any parameters defining the basis is an "ab initio" way to define them.

We have also devised a quite simple and systematic way of generating basis sets based on specifying only one main parameter (the energy shift) besides the basis size. It does not offer the best NAO results one can get for a given basis size but it has the important advantages mentioned above. More about it in:

"Linear-scaling ab-initio calculations for large and complex systems", E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

In addition to SIESTA we provide the program `Gen-basis` , which reads SIESTA's input and generates basis files for later use. `Gen-basis` can be found in `Util/Gen-basis`. It should be run from the `Tutorials/Bases` directory, using the `gen-basis.sh` script. It is limited to a single species.

Of course, as it happens for the pseudopotential, it is the responsibility of the user to check that the physical results obtained are converged with respect to the basis set used before starting any production run.

In the following we give some clues on the basics of the basis sets that SIESTA generates. The starting point is always the solution of Kohn-Sham's Hamiltonian for the isolated pseudo-atoms, solved in a radial grid, with the same approximations as for the solid or molecule (the same exchange-correlation functional and pseudopotential), plus some way of confinement (see below). We describe in the following three main features of a basis set of atomic orbitals: size, range, and radial shape.

Size: number of orbitals per atom

Following the nomenclature of Quantum Chemistry, we establish a hierarchy of basis sets, from single-ζ to multiple-ζ with polarization and diffuse orbitals, covering from quick calculations of low quality to high precision, as high as the finest obtained in Quantum Chemistry. A single-ζ (also called minimal) basis set (SZ in the following) has one single radial function per angular momentum channel, and only for those angular momenta with substantial electronic population in the valence of the free atom. It offers quick calculations and some insight on qualitative trends in the chemical bonding and other properties. It remains too rigid, however, for more quantitative calculations

requiring both radial and angular flexibilization.

Starting by the radial flexibilization of SZ, a better basis is obtained by adding a second function per channel: double-ζ (DZ). In Quantum Chemistry, the *split valence* scheme is widely used: starting from the expansion in Gaussians of one atomic orbital, the most contracted Gaussians are used to define the first orbital of the double-ζ and the most extended ones for the second. For strictly localized functions there was a first proposal of using the excited states of the confined atoms, but it would work only for tight confinement (see **PAO.BasisType** `nodes` below). This construction was proposed and tested in D. Sánchez-Portal *et al.*, J. Phys.: Condens. Matter **8**, 3859-3880 (1996).

We found that the basis set convergence is slow, requiring high levels of multiple-ζ to achieve what other schemes do at the double-ζ level. This scheme is related with the basis sets used in the OpenMX project [see T. Ozaki, Phys. Rev. B **67**, 155108 (2003); T. Ozaki and H. Kino, Phys. Rev. B **69**, 195113 (2004)].

We then proposed an extension of the split valence idea of Quantum Chemistry to strictly localized NAO which has become the standard and has been used quite successfully in many systems (see **PAO.BasisType** `split` below). It is based on the idea of suplementing the first ζ with, instead of a gaussian, a numerical orbital that reproduces the tail of the original PAO outside a matching radius r_m, and continues smoothly towards the origin as $r^l(a - br^2)$, with a and b ensuring continuity and differentiability at r_m. Within exactly the same Hilbert space, the second orbital can be chosen to be the difference between the smooth one and the original PAO, which gives a basis orbital strictly confined within the matching radius r_m (smaller than the original PAO!) continuously differentiable throughout.

Extra parameters have thus appeared: one r_m per orbital to be doubled. The user can again introduce them by hand (see **PAO.Basis** below). Alternatively, all the r_m's can be defined at once by specifying the value of the tail of the original PAO beyond r_m, the so-called split norm. Variational optimization of this split norm performed on different systems shows a very general and stable performance for values around 15% (except for the $\sim$ 50% for hydrogen). It generalizes to multiple-ζ trivially by adding an additional matching radius per new zeta.

Note: What is actually used is the norm of the tail *plus* the norm of the parabola-like inner function.

Angular flexibility is obtained by adding shells of higher angular momentum. Ways to generate these so-called polarization orbitals have been described in the literature for Gaussians. For NAOs there are two ways for SIESTA and `Gen-basis` to generate them: (*i*) Use atomic PAO's of higher angular momentum with suitable confinement, and (*ii*) solve the pseudoatom in the presence of an electric field and obtain the $l + 1$ orbitals from the perturbation of the l orbitals by the field.

So-called diffuse orbitals, that might be important in the description of open systems such as surfaces, can be simply added by specifying extra "n" shells. [See S. Garcia-Gil, A. Garcia, N. Lorente, P. Ordejon, Phys. Rev. B **79**, 075441 (2009)]

Finally, the method allows the inclusion of off-site (ghost) orbitals (not centered around any specific atom), useful for example in the calculation of the counterpoise correction for basis-set superposition errors. Bessel functions for any radius and any excitation level can also be added anywhere to the basis set.

Range: cutoff radii of orbitals.

Strictly localized orbitals (zero beyond a cutoff radius) are used in order to obtain sparse Hamiltonian and overlap matrices for linear scaling. One cutoff radius per angular momentum channel has to be

given for each species.

A balanced and systematic starting point for defining all the different radii is achieved by giving one single parameter, the energy shift, i.e., the energy increase experienced by the orbital when confined. Allowing for system and physical-quantity variablity, as a rule of thumb $\Delta E_{\mathrm{PAO}} \approx 100$ meV gives typical precisions within the accuracy of current GGA functionals. The user can, nevertheless, change the cutoff radii at will.

Shape

Within the pseudopotential framework it is important to keep the consistency between the pseudopotential and the form of the pseudoatomic orbitals in the core region. The shape of the orbitals at larger radii depends on the cutoff radius (see above) and on the way the localization is enforced.

The first proposal (and quite a standard among SIESTA users) uses an infinite square-well potential. It was originally proposed and has been widely and successfully used by Otto Sankey and collaborators, for minimal bases within the ab initio tight-binding scheme, using the `Fireball` program, but also for more flexible bases using the methodology of SIESTA. This scheme has the disadavantage, however, of generating orbitals with a discontinuous derivative at r_c. This discontinuity is more pronounced for smaller r_c's and tends to disappear for long enough values of this cutoff. It does remain, however, appreciable for sensible values of r_c for those orbitals that would be very wide in the free atom. It is surprising how small an effect such a kink produces in the total energy of condensed systems. It is, on the other hand, a problem for forces and stresses, especially if they are calculated using a (coarse) finite three-dimensional grid.

Another problem of this scheme is related to its defining the basis starting from the free atoms. Free atoms can present extremely extended orbitals, their extension being, besides problematic, of no practical use for the calculation in condensed systems: the electrons far away from the atom can be described by the basis functions of other atoms.

A traditional scheme to deal with this is one based on the radial scaling of the orbitals by suitable scale factors. In addition to very basic bonding arguments, it is soundly based on restoring the virial's theorem for finite bases, in the case of Coulombic potentials (all-electron calculations). The use of pseudopotentials limits its applicability, allowing only for extremely small deviations from unity ($\sim 1\%$) in the scale factors obtained variationally (with the exception of hydrogen that can contract up to 25%). This possiblity is available to the user.

Another way of dealing with the above problem and that of the kink at the same time is adding a soft confinement potential to the atomic Hamiltonian used to generate the basis orbitals: it smoothens the kink and contracts the orbital as suited. Two additional parameters are introduced for the purpose, which can be defined again variationally. The confining potential is flat (zero) in the core region, starts off at some internal radius r_i with all derivatives continuous and diverges at r_c ensuring the strict localization there. It is

$$V(r) = V_{\mathrm{o}} \frac{e^{-\frac{r_c - r_i}{r - r_i}}}{r_c - r} \tag{1}$$

and both r_i and V_{o} can be given to SIESTA together with r_c in the input (see **PAO.Basis** below). The kink is normally well smoothened with the default values for soft confinement by default (**PAO.SoftDefault** true), which are $r_i = 0.9 r_c$ and $V_{\mathrm{o}} = 40\,\mathrm{Ry}$.

When explicitly introducing orbitals in the basis that would be empty in the atom (e.g. polarisation orbitals) these tend to be extremely extended if not completely unbound. The above procedure produces orbitals that bulge as far away from the nucleus as possible, to plunge abruptly at r_c. Soft

confinement can be used to try to force a more reasonable shape, but it is not ideal (for orbitals peaking in the right region the tails tend to be far too short). *Charge confinement* produces very good shapes for empty orbitals. Essentially a Z/r potential is added to the soft confined potential above. For flexibility the charge confinement option in SIESTA is defined as

$$V_{\mathrm{Q}}(r) = \frac{Ze^{-\lambda r}}{\sqrt{r^2+\delta^2}} \tag{2}$$

where δ is there to avoid the singularity (default $\delta = 0.01$ Bohr), and λ allows to screen the potential if longer tails are needed. The description on how to introduce this option can be found in the **PAO.Basis** entry below.

Finally, the shape of an orbital is also changed by the ionic character of the atom. Orbitals in cations tend to shrink, and they swell in anions. Introducing a δQ in the basis-generating free-atom calculations gives orbitals better adapted to ionic situations in the condensed systems.

More information about basis sets can be found in the proposed literature.

There are quite a number of options for the input of the basis-set and KB projector specification, and they are all optional! By default, SIESTA will use a DZP basis set with appropriate choices for the determination of the range, etc. Of course, the more you experiment with the different options, the better your basis set can get. To aid in this process we offer an auxiliary program for optimization which can be used in particular to obtain variationally optimal basis sets (within a chosen basis size). See `Util/Optimizer` for general information, and `Util/Optimizer/Examples/Basis_Optim` for an example. The directory `Tutorials/Bases` in the main SIESTA distribution contains some tutorial material for the generation of basis sets and KB projectors.

Finally, some optimized basis sets for particular elements are available at the SIESTA web page. Again, it is the responsability of the users to test the transferability of the basis set to their problem under consideration.

6.3.2 Type of basis sets

PAO.BasisType **split** *(string)*

The kind of basis to be generated is chosen. All are based on finite-range pseudo-atomic orbitals [PAO's of Sankey and Niklewsky, PRB 40, 3979 (1989)]. The original PAO's were described only for minimal bases. SIESTA generates extended bases (multiple-ζ, polarization, and diffuse orbitals) applying different schemes of choice:

- Generalization of the PAO's: uses the excited orbitals of the finite-range pseudo-atomic problem, both for multiple-ζ and for polarization [see Sánchez-Portal, Artacho, and Soler, JPCM **8**, 3859 (1996)]. Adequate for short-range orbitals.
- Multiple-ζ in the spirit of split valence, decomposing the original PAO in several pieces of different range, either defining more (and smaller) confining radii, or introducing Gaussians from known bases (Huzinaga's book).

All the remaining options give the same minimal basis. The different options and their FDF descriptors are the following:

split Split-valence scheme for multiple-zeta. The split is based on different radii.

splitgauss Same as `split` but using gaussian functions $e^{-(x/\alpha_i)^2}$. The gaussian widths α_i are read instead of the scale factors (see below). There is no cutting algorithm, so that a large enough r_c should be defined for the gaussian to have decayed sufficiently.

nodes Generalized PAO's.

nonodes The original PAO's are used, multiple-zeta is generated by changing the scale-factors, instead of using the excited orbitals.

filteret Use the filterets as a systematic basis set. The size of the basis set is controlled by the filter cut-off for the orbitals.

Note that, for the **split** and **nodes** cases the whole basis can be generated by SIESTA with no further information required. SIESTA will use default values as defined in the following (**PAO.BasisSize**, **PAO.EnergyShift**, and **PAO.SplitNorm**, see below).

6.3.3 Size of the basis set

PAO.BasisSize DZP *(string)*

It defines usual basis sizes. It has effect only if there is no block **PAO.Basis** present.

SZ|minimal Use single-ζ basis.

DZ Double zeta basis, in the scheme defined by **PAO.BasisType**.

SZP Single-zeta basis plus polarization orbitals.

DZP|standard Like **DZ** plus polarization orbitals. Polarization orbitals are constructed from perturbation theory, and they are defined so they have the minimum angular momentum l such that there are not occupied orbitals with the same l in the valence shell of the ground-state atomic configuration. They polarize the corresponding $l-1$ shell.

NOTE: the ground-state atomic configuration used internally by SIESTA is defined in the source file `Src/periodic_table.f`. For some elements (e.g., Pd), the configuration might not be the standard one.

%block PAO.BasisSizes ⟨None⟩ *(block)*

Block which allows to specify a different value of the variable **PAO.BasisSize** for each species. For example,

```
%block PAO.BasisSizes
    Si      DZ
    H       DZP
    O       SZP
%endblock PAO.BasisSizes
```

6.3.4 Range of the orbitals

PAO.EnergyShift 0.02 Ry *(energy)*

A standard for orbital-confining cutoff radii. It is the excitation energy of the PAO's due to the confinement to a finite-range. It offers a general procedure for defining the confining radii of the original (first-zeta) PAO's for all the species guaranteeing the compensation of the basis. It only has an effect when the block **PAO.Basis** is not present or when the radii specified in

that block are zero for the first zeta.

Write.Graphviz **none|atom|orbital|atom+orbital** *(string)*

Write out the sparsity pattern after having determined the basis size overlaps. This will generate `SystemLabel.ATOM.gv` or `SystemLabel.ORB.gv` which both may be converted to a graph using Graphviz's program `neato`:

```
neato -x -Tpng siesta.ATOM.gv -o siesta_ATOM.png
```

The resulting graph will list each atom as $i(j)$ where i is the atomic index and j is the number of other atoms it is connected to.

6.3.5 Generation of multiple-zeta orbitals

PAO.SplitNorm 0.15 *(real)*

A standard to define sensible default radii for the split-valence type of basis. It gives the amount of norm that the second-ζ split-off piece has to carry. The split radius is defined accordingly. If multiple-ζ is used, the corresponding radii are obtained by imposing smaller fractions of the SplitNorm (1/2, 1/4 ...) value as norm carried by the higher zetas. It only has an effect when the block **PAO.Basis** is not present or when the radii specified in that block are zero for zetas higher than one.

PAO.SplitNormH ⟨**PAO.SplitNorm**⟩ *(real)*

This option is as per **PAO.SplitNorm** but allows a separate default to be specified for hydrogen which typically needs larger values than those for other elements.

PAO.NewSplitCode **false** *(logical)*

Enables a new, simpler way to match the multiple-zeta radii.

If an old-style (tail+parabola) calculation is being done, perform a scan of the tail+parabola norm in the whole range of the 1st-zeta orbital, and store that in a table. The construction of the 2nd-zeta orbital involves simply scanning the table to find the appropriate place. Due to the idiosyncracies of the old algorithm, the new one is not guaranteed to produce exactly the same results, as it might settle on a neighboring grid point for the matching.

PAO.FixSplitTable **false** *(logical)*

After the scan of the allowable split-norm values, apply a damping function to the tail to make sure that the table goes to zero at the radius of the first-zeta orbital.

PAO.SplitTailNorm **false** *(logical)*

Use the norm of the tail instead of the full tail+parabola norm. This is the behavior described in the JPC paper. (But note that, for numerical reasons, the square root of the tail norm is used in the algorithm.) This is the preferred mode of operation for automatic operation, as in non-supervised basis-optimization runs.

As a summary of the above options:

- For complete backwards compatibility, do nothing.
- To exercise the new code, set **PAO.NewSplitCode**.

- To maintain the old split-norm heuristic, but making sure that the program finds a solution (even if not optimal, in the sense of producing a second-ζ r_c very close to the first-ζ one), set **PAO.FixSplitTable** (this will automatically set **PAO.NewSplitCode**).
- If the old heuristic is of no interest (for example, if only a robust way of mapping split-norms to radii is needed), set **PAO.SplitTailNorm** (this will set **PAO.NewSplitCode** automatically).

PAO.EnergyCutoff 20 Ry *(energy)*

If the multiple zetas are generated using filterets then only the filterets with an energy lower than this cutoff are included. Increasing this value leads to a richer basis set (provided the cutoff is raised above the energy of any filteret that was previously not included) but a more expensive calculation. It only has an effect when the option **PAO.BasisType** is set to **filteret**.

PAO.EnergyPolCutoff 20 Ry *(energy)*

If the multiple zetas are generated using filterets then only the filterets with an energy lower than this cutoff are included for the polarisation functions. Increasing this value leads to a richer basis set (provided the cutoff is raised above the energy of any filteret that was previously not included) but a more expensive calculation. It only has an effect when the option **PAO.BasisType** is set to filteret.

PAO.ContractionCutoff $0|0-1$ *(real)*

If the multiple zetas are generated using filterets then any filterets that have a coefficient less than this threshold within the original PAO will be contracted together to form a single filteret. Increasing this value leads to a smaller basis set but allows the underlying basis to have a higher kinetic energy cut-off for filtering. It only has an effect when the option **PAO.BasisType** is set to **filteret**.

6.3.6 Soft-confinement options

PAO.SoftDefault **false** *(logical)*

If set to true then this option causes soft confinement to be the default form of potential during orbital generation. The default potential and inner radius are set by the commands given below.

PAO.SoftInnerRadius 0.9 *(real)*

For default soft confinement, the inner radius is set at a fraction of the outer confinement radius determined by the energy shift. This option controls the fraction of the confinement radius to be used.

PAO.SoftPotential 40 Ry *(energy)*

For default soft confinement, this option controls the value of the potential used for all orbitals.

NOTE: Soft-confinement options (inner radius, prefactor) have been traditionally used to optimize the basis set, even though formally they are just a technical necessity to soften the decay of the orbitals at rc. To achieve this, it might be enough to use the above global options.

6.3.7 Kleinman-Bylander projectors

%block PS.lmax ⟨**None**⟩ *(block)*

Block with the maximum angular momentum of the Kleinman-Bylander projectors, `lmxkb`. This information is optional. If the block is absent, or for a species which is not mentioned inside it, SIESTA will take `lmxkb(is) = lmxo(is) + 1`, where `lmxo(is)` is the maximum angular momentum of the basis orbitals of species `is`.

```
%block Ps.lmax
    Al_adatom   3
    H           1
    O           2
%endblock Ps.lmax
```

By default `lmax` is the maximum angular momentum plus one.

%block PS.KBprojectors ⟨**None**⟩ *(block)*

This block provides information about the number of Kleinman-Bylander projectors per angular momentum, and for each species, that will used in the calculation. This block is optional. If the block is absent, or for species not mentioned in it, only one projector will be used for each angular momentum. The projectors will be constructed using the eigenfunctions of the respective pseudopotentials.

This block allows to specify the number of projector for each l, and also the reference energies of the wavefunctions used to build them. The specification of the reference energies is optional. If these energies are not given, the program will use the eigenfunctions with an increasing number of nodes (if there is not bound state with the corresponding number of nodes, the "eigenstates" are taken to be just functions which are made zero at very long distance of the nucleus). The units for the energy can be optionally specified, if not, the program will assumed that are given in Rydbergs. The data provided in this block must be consistent with those read from the block **PS.lmax**. For example,

```
%block PS.KBprojectors
    Si  3
     2   1
    -0.9     eV
     0   2
    -0.5  -1.0d4 Hartree
     1   2
    Ga  1
     1  3
    -1.0  1.0d5 -6.0
%endblock PS.KBprojectors
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present):

```
From is = 1 to  nspecies
    read: label(is), l_shells(is)
    From lsh=1 to l_shells(is)
        read: l, nkbl(l,is)
        read: {erefKB(izeta,il,is)}, from ikb = 1 to nkbl(l,is), {units}
```

When a very high energy, higher that 1000 Ry, is specified, the default is taken instead. On the other hand, very low (negative) energies, lower than -1000 Ry, are used to indicate that

the energy derivative of the last state must be used. For example, in the example given above, two projectors will be used for the *s* pseudopotential of Si. One generated using a reference energy of -0.5 Hartree, and the second one using the energy derivative of this state. For the *p* pseudopotential of Ga, three projectors will be used. The second one will be constructed from an automatically generated wavefunction with one node, and the other projectors from states at -1.0 and -6.0 Rydberg.

The analysis looking for possible *ghost* states is only performed when a single projector is used. Using several projectors some attention should be paid to the "KB cosine" (kbcos), given in the output of the program. The KB cosine gives the value of the overlap between the reference state and the projector generated from it. If these numbers are very small (< 0.01, for example) for **all** the projectors of some angular momentum, one can have problems related with the presence of ghost states.

The default is *one* KB projector from each angular momentum, constructed from the nodeless eigenfunction.

KB.New.Reference.Orbitals **false** *(logical)*

If **true**, the routine to generate KB projectors will use slightly different parameters for the construction of the reference orbitals involved (Rmax=60 Bohr both for integration and normalization).

6.3.8 The PAO.Basis block

%block PAO.Basis ⟨**None**⟩ *(block)*

Block with data to define explicitly the basis to be used. It allows the definition by hand of all the parameters that are used to construct the atomic basis. There is no need to enter information for all the species present in the calculation. The basis for the species not mentioned in this block will be generated automatically using the parameters **PAO.BasisSize**, **PAO.BasisType**, **PAO.EnergyShift**, **PAO.SplitNorm** (or **PAO.SplitNormH**), and the soft-confinement defaults, if used (see **PAO.SoftDefault**).

Some parameters can be set to zero, or left out completely. In these cases the values will be generated from the magnitudes defined above, or from the appropriate default values. For example, the radii will be obtained from **PAO.EnergyShift** or from **PAO.SplitNorm** if they are zero; the scale factors will be put to 1 if they are zero or not given in the input. An example block for a two-species calculation (H and O) is the following (`opt` means optional):

```
%block PAO.Basis     # Define Basis set
O    2  nodes  1.0   # Label, l_shells, type (opt), ionic_charge (opt)
 n=2 0 2  E 50.0 2.5 # n (opt if not using semicore levels),l,Nzeta,Softconf(opt)
     3.50  3.50      #     rc(izeta=1,Nzeta)(Bohr)
     0.95  1.00      #     scaleFactor(izeta=1,Nzeta) (opt)
     1 1  P 2        # l, Nzeta, PolOrb (opt), NzetaPol (opt)
     3.50            #     rc(izeta=1,Nzeta)(Bohr)
H    2               # Label, l_shells, type (opt), ionic_charge (opt)
     0 2 S 0.2       # l, Nzeta, Per-shell split norm parameter
     5.00  0.00      #     rc(izeta=1,Nzeta)(Bohr)
     1 1 Q 3. 0.2    # l, Nzeta, Charge conf (opt): Z and screening
     5.00            #    rc(izeta=1,Nzeta)(Bohr)
%endblock PAO.Basis
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present) (See the routines in `Src/basis_specs.f` for detailed information):

```
From js = 1 to  nspecies
   read: label(is), l_shells(is), { type(is) }, { ionic_charge(is) }
   From lsh=1 to l_shells(is)
    read:
     { n }, l(lsh), nzls(lsh,is), { PolOrb(l+1) }, { NzetaPol(l+1) },
     {SplitNormfFlag(lsh,is)}, {SplitNormValue(lsh,is)}
     {SoftConfFlag(lsh,is)}, {PrefactorSoft(lsh,is)}, {InnerRadSoft(lsh,is)},
     {FilteretFlag(lsh,is)}, {FilteretCutoff(lsh,is)}
     {ChargeConfFlag(lsh,is)}, {Z(lsh,is)}, {Screen(lsh,is)}, {delta(lsh,is}
       read: rcls(izeta,lsh,is), from izeta = 1 to nzls(l,is)
       read: { contrf(izeta,il,is) }, from izeta = 1 to nzls(l,is)
```

And here is the variable description:

- `Label`: Species label, this label determines the species index `is` according to the block **ChemicalSpeciesLabel**
- `l_shells(is)`: Number of shells of orbitals with different angular momentum for species `is`
- `type(is)`: *Optional input*. Kind of basis set generation procedure for species `is`. Same options as **PAO.BasisType**
- `ionic_charge(is)`: *Optional input*. Net charge of species `is`. This is only used for basis set generation purposes. *Default value*: `0.0` (neutral atom). Note that if the pseudopotential was generated in an ionic configuration, and no charge is specified in PAO.Basis, the ionic charge setting will be that of pseudopotential generation.
- `n`: Principal quantum number of the shell. This is an optional input for normal atoms, however it must be specified when there are *semicore* states (i.e. when states that usually are not considered to belong to the valence shell have been included in the calculation)
- `l`: Angular momentum of basis orbitals of this shell
- `nzls(lsh,is)`: Number of "zetas" for this shell. For a filteret basis this number is ignored since the number is controlled by the cutoff.
- `PolOrb(l+1)`: *Optional input*. If set equal to `P`, a shell of polarization functions (with angular momentum $l+1$) will be constructed from the first-zeta orbital of angular momentum l. *Default value*: ' ' (blank = No polarization orbitals).
- `NzetaPol(l+1)`: *Optional input*. Number of "zetas" for the polarization shell (generated automatically in a split-valence fashion). For a filteret basis this number is ignored since the number is controlled by the cutoff. Only active if `PolOrb = P`. *Default value*: `1`
- `SplitNormFlag(lsh,is)`: *Optional input*. If set equal to `S`, the following number sets the split-norm parameter for that shell.
- `SoftConfFlag(l,is)`: *Optional input*. If set equal to `E`, the soft confinement potential proposed in equation (1) of the paper by J. Junquera *et al.*, Phys. Rev. B **64**, 235111 (2001), is used instead of the Sankey hard-well potential.
- `PrefactorSoft(l,is)`: *Optional input*. Prefactor of the soft confinement potential (V_0 in the formula). Units in Ry. *Default value*: 0 Ry.

- `InnerRadSoft(l,is)`: *Optional input.* Inner radius where the soft confinement potential starts off (r_i in the formula). If negative, the inner radius will be computed as the given fraction of the PAO cutoff radius. Units in bohrs. *Default value*: 0 bohrs.
- `FilteretFlag(l,is)`: *Optional input.* If set equal to `F`, then an individual filter cut-off can be specified for the shell.
- `FilteretCutoff(l,is)`: *Optional input.* Shell-specific value for the filteret basis cutoff. Units in Ry. *Default value*: The same as the value given by **FilterCutoff**.
- `ChargeConfFlag(lsh,is)`: *Optional input.* If set equal to `Q`, the charge confinement potential in equation (2) above is added to the confining potential. If present it requires at least one number after it (`Z`), but it can be followed by two or three numbers.
- `Z(lhs,is)`: *Optional input, needed if `Q` is set.* Z charge in equation (2) above for charge confinement (units of e).
- `Screen(lhs,is)`: *Optional input.* Yukawa screening parameter λ in equation (2) above for charge confinement (in Bohr^{-1}).
- `delta(lhs,is)`: *Optional input.* Singularity regularisation parameter δ in equation (2) above for charge confinement (in Bohr).
- `rcls(izeta,l,is)`: Cutoff radius (Bohr) of each 'zeta' for this shell. For the second zeta onwards, if this value is negative, the actual rc used will be the given fraction of the first zeta's rc.
- `contrf(izeta,l,is)`: *Optional input.* Contraction factor of each "zeta" for this shell. *Default value*: `1.0`

Polarization orbitals are generated by solving the atomic problem in the presence of a polarizing electric field. The orbitals are generated applying perturbation theory to the first-zeta orbital of lower angular momentum. They have the same cutoff radius as the orbitals from which they are constructed.

Note: The perturbative method has traditionally used the 'l' component of the pseudopotential. It can be argued that it should use the 'l+1' component. By default, for backwards compatibility, the traditional method is used, but the alternative one can be activated by setting the logical **PAO.OldStylePolOrbs** variable to **false**.

There is a different possibility for generating polarization orbitals: by introducing them explicitly in the **PAO.Basis** block. It has to be remembered, however, that they sometimes correspond to unbound states of the atom, their shape depending very much on the cutoff radius, not converging by increasing it, similarly to the multiple-zeta orbitals generated with the `nodes` option. Using **PAO.EnergyShift** makes no sense, and a cut off radius different from zero must be explicitly given (the same cutoff radius as the orbitals they polarize is usually a sensible choice).

A species with atomic number = -100 will be considered by SIESTA as a constant-pseudopotential atom, *i.e.*, the basis functions generated will be spherical Bessel functions with the specified r_c. In this case, r_c has to be given, as **EnergyShift** will not calculate it.

Other negative atomic numbers will be interpreted by SIESTA as *ghosts* of the corresponding positive value: the orbitals are generated and put in position as determined by the coordinates, but neither pseudopotential nor electrons are considered for that ghost atom. Useful for BSSE correction.

Use: This block is optional, except when Bessel functions or semicore states are present.

Default: Basis characteristics defined by global definitions given above.

6.3.9 Filtering

FilterCutoff 0 eV *(energy)*

Kinetic energy cutoff of plane waves used to filter all the atomic basis functions, the pseudo-core densities for partial core corrections, and the neutral-atom potentials. The basis functions (which must be squared to obtain the valence density) are really filtered with a cutoff reduced by an empirical factor $0.7^2 \simeq 0.5$. The **FilterCutoff** should be similar or lower than the **MeshCutoff** to avoid the *eggbox effect* on the atomic forces. However, one should not try to converge **MeshCutoff** while simultaneously changing **FilterCutoff**, since the latter in fact changes the used basis functions. Rather, fix a sufficiently large **FilterCutoff** and converge only **MeshCutoff**. If **FilterCutoff** is not explicitly set, its value is calculated from **FilterTol**.

FilterTol 0 eV *(energy)*

Residual kinetic-energy leaked by filtering each basis function. While **FilterCutoff** sets a common reciprocal-space cutoff for all the basis functions, **FilterTol** sets a specific cutoff for each basis function, much as the **PAO.EnergyShift** sets their real-space cutoff. Therefore, it is reasonable to use similar values for both parameters. The maximum cutoff required to meet the **FilterTol**, among all the basis functions, is used (multiplied by the empirical factor $1/0.7^2 \simeq 2$) to filter the pseudo-core densities and the neutral-atom potentials. **FilterTol** is ignored if **FilterCutoff** is present in the input file. If neither **FilterCutoff** nor **FilterTol** are present, no filtering is performed. See Soler and Anglada, arXiv:0807.5030, for details of the filtering procedure.

Warning: If the value of **FilterCutoff** is made too small (or **FilterTol** too large) some of the filtered basis orbitals may be meaningless, leading to incorrect results or even a program crash.

To be implemented: If **MeshCutoff** is not present in the input file, it can be set using the maximum filtering cutoff used for the given **FilterTol** (for the time being, you can use **AtomSetupOnly true** to stop the program after basis generation, look at the maximum filtering cutoff used, and set the mesh-cutoff manually in a later run.)

6.3.10 Saving and reading basis-set information

SIESTA (and the standalone program Gen-basis) always generate the files *Atomlabel*`.ion`, where *Atomlabel* is the atomic label specified in block **ChemicalSpeciesLabel**. Optionally, if NetCDF support is compiled in, the programs generate NetCDF files *Atomlabel*`.ion.nc` (except for ghost atoms). See an Appendix for information on the optional NetCDF package.

These files can be used to read back information into SIESTA.

User.Basis **false** *(logical)*

If true, the basis, KB projector, and other information is read from files *Atomlabel*`.ion`, where *Atomlabel* is the atomic species label specified in block **ChemicalSpeciesLabel**. These files can be generated by a previous SIESTA run or (one by one) by the standalone program `Gen-basis`. No pseudopotential files are necessary.

User.Basis.NetCDF **false** *(logical)*

If true, the basis, KB projector, and other information is read from NetCDF files *Atomlabel*`.ion.nc`, where *Atomlabel* is the atomic label specified in block **ChemicalSpeciesLabel**. These files can be generated by a previous SIESTA run or by the standalone program `Gen-basis`. No pseudopotential files are necessary. NetCDF support is needed. Note that ghost atoms cannot yet be adequately treated with this option.

6.3.11 Tools to inspect the orbitals and KB projectors

The program `ioncat` in `Util/Gen-basis` can be used to extract orbital, KB projector, and other information contained in the `.ion` files. The output can be easily plotted with a graphics program. If the option **WriteIonPlotFiles** is enabled, SIESTA will generate and extra set of files that can be plotted with the `gnuplot` scripts in `Tutorials/Bases`. The stand-alone program `gen-basis` sets that option by default, and the script `Tutorials/Bases/gen-basis.sh` can be used to automate the process. See also the NetCDF-based utilities in `Util/PyAtom`.

6.3.12 Basis optimization

There are quite a number of options for the input of the basis-set and KB projector specification, and they are all optional! By default, SIESTA will use a DZP basis set with appropriate choices for the determination of the range, etc. Of course, the more you experiment with the different options, the better your basis set can get. To aid in this process we offer an auxiliary program for optimization which can be used in particular to obtain variationally optimal basis sets (within a chosen basis size). See `Util/Optimizer` for general information, and `Util/Optimizer/Examples/Basis_Optim` for an example.

BasisPressure 0.2 GPa *(pressure)*

SIESTA will compute and print the value of the "effective basis enthalpy" constructed by adding a term of the form $p_{basis}V_{orbs}$ to the total energy. Here p_{basis} is a fictitious basis pressure and V_{orbs} is the volume of the system's orbitals. This is a useful quantity for basis optimization (See Anglada *et al.*). The total basis enthalpy is also written to the ASCII file `BASIS_ENTHALPY`.

6.3.13 Low-level options regarding the radial grid

For historical reasons, the basis-set and KB projector code in SIESTA uses a logarithmic radial grid, which is taken from the pseudopotential file. Any "interesting" radii have to fall on a grid point, which introduces a certain degree of coarseness that can limit the accuracy of the results and the faithfulness of the mapping of input parameters to actual operating parameters. For example, the same orbital will be produced by a finite range of **PAO.EnergyShift** values, and any user-defined cutoffs will not be exactly reflected in the actual cutoffs. This is particularly troublesome for automatic optimization procedures (such as those implemented in `Util/Optimizer`), as the engine might be confused by the extra level of indirection. The following options can be used to fine-tune the mapping. They are not enabled by default, as they change the numerical results apreciably (in effect, they lead to different basis orbitals and projectors).

Reparametrize.Pseudos **false** *(logical)*

By changing the a and b parameters of the logarithmic grid, a new one with a more adequate

grid-point separation can be used for the generation of basis sets and projectors. For example, by using $a = 0.001$ and $b = 0.01$, the grid point separations at $r = 0$ and 10 bohrs are 0.00001 and 0.01 bohrs, respectively. More points are needed to reach r's of the order of a hundred bohrs, but the extra computational effort is negligible. The net effect of this option (notably when coupled to **Restricted.Radial.Grid false**) is a closer mapping of any user-specified cutoff radii and of the radii implicitly resulting from other input parameters to the actual values used by the program. (The small grid-point separation near r=0 is still needed to avoid instabilities for s channels that occurred with the previous (reparametrized) default spacing of 0.005 bohr. This effect is not yet completely understood.)

New.A.Parameter 0.001 *(real)*

New setting for the pseudopotential grid's a parameter

New.B.Parameter 0.01 *(real)*

New setting for the pseudopotential grid's b parameter

Rmax.Radial.Grid 50.0 *(real)*

New setting for the maximum value of the radial coordinate for integration of the atomic Schrodinger equation.

If **Reparametrize.Pseudos** is **false** this will be the maximum radius in the pseudopotential file.

Restricted.Radial.Grid **true** *(logical)*

In normal operation of the basis-set and projector generation code the various cutoff radii are restricted to falling on an odd-numbered grid point, shifting then accordingly. This restriction can be lifted by setting this parameter to **false**.

6.4 Structural information

There are many ways to give SIESTA structural information.

- Directly from the fdf file in traditional format.
- Directly from the fdf file in the newer Z-Matrix format, using a **Zmatrix** block.
- From an external data file

Note that, regardless of the way in which the structure is described, the **ChemicalSpeciesLabel** block is mandatory.

In the following sections we document the different structure input methods, and provide a guide to their precedence.

6.4.1 Traditional structure input in the fdf file

Firstly, the size of the cell itself should be specified, using some combination of the options **LatticeConstant**, **LatticeParameters**, and **LatticeVectors**, and **SuperCell**. If nothing is specified, SIESTA will construct a cubic cell in which the atoms will reside as a cluster.

Secondly, the positions of the atoms within the cells must be specified, using either the traditional SIESTA input format (a modified xyz format) which must be described within a **AtomicCoordinatesAndAtomicSpecies** block.

LatticeConstant ⟨**None**⟩ *(length)*

Lattice constant. This is just to define the scale of the lattice vectors.

Default value: Minimum size to include the system (assumed to be a molecule) without intercell interactions, plus 10%.

NOTE: A LatticeConstant value, even if redundant, might be needed for other options, such as the units of the k-points used for band-structure calculations. This mis-feature will be corrected in future versions.

%block LatticeParameters ⟨**None**⟩ *(block)*

Crystallographic way of specifying the lattice vectors, by giving six real numbers: the three vector modules, a, b, and c, and the three angles α (angle between $\vec{b}$ and $\vec{c}$), β, and γ. The three modules are in units of **LatticeConstant**, the three angles are in degrees.

This defaults to a square cell with side-lengths equal to **LatticeConstant**.

```
1.0  1.0  1.0   90.  90.  90.
```

%block LatticeVectors ⟨**None**⟩ *(block)*

The cell vectors are read in units of the lattice constant defined above. They are read as a matrix `CELL(ixyz,ivector)`, each vector being one line.

This defaults to a square cell with side-lengths equal to **LatticeConstant**.

```
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0
```

If the **LatticeConstant** default is used, the default of **LatticeVectors** is still diagonal but not necessarily cubic.

%block SuperCell ⟨**None**⟩ *(block)*

Integer 3x3 matrix defining a supercell in terms of the unit cell. Any values larger than 1 will expand the unitcell (plus atoms) along that lattice vector direction (if possible).

```
%block SuperCell
   M(1,1)  M(2,1)  M(3,1)
   M(1,2)  M(2,2)  M(3,2)
   M(1,3)  M(2,3)  M(3,3)
%endblock SuperCell
```

and the supercell is defined as $\mathrm{SuperCell}(ix, i) = \sum_j \mathrm{CELL}(ix, j) * M(j, i)$. Notice that the matrix indexes are inverted: each input line specifies one supercell vector.

Warning: **SuperCell** is disregarded if the geometry is read from the XV file, which can happen inadvertently.

Use: The atomic positions must be given only for the unit cell, and they are 'cloned' automatically in the rest of the supercell. The **NumberOfAtoms** given must also be that in a single unit cell. However, all values in the output are given for the entire supercell. In fact, `CELL` is immediately redefined as the whole supercell and the program no longer knows the existence of an underlying unit cell. All other input (apart from NumberOfAtoms and atomic positions),

including **kgrid.MonkhorstPack** must refer to the supercell (this is a change over previous versions). Therefore, to avoid confusions, we recommend to use **SuperCell** only to generate atomic positions, and then to copy them from the output to a new input file with all the atoms specified explicitly and with the supercell given as a normal unit cell.

AtomicCoordinatesFormat **Bohr** *(string)*

Character string to specify the format of the atomic positions in input. These can be expressed in four forms:

Bohr|NotScaledCartesianBohr atomic positions are given directly in Bohr, in Cartesian coordinates

Ang|NotScaledCartesianAng atomic positions are given directly in Ångström, in Cartesian coordinates

ScaledCartesian atomic positions are given in Cartesian coordinates, in units of the lattice constant

Fractional|ScaledByLatticeVectors atomic positions are given referred to the lattice vectors

AtomCoorFormatOut ⟨**AtomicCoordinatesFormat**⟩ *(string)*

Character string to specify the format of the atomic positions in output.

Same possibilities as for input **AtomicCoordinatesFormat**.

%block AtomicCoordinatesOrigin ⟨**None**⟩ *(block)*

Vector specifying a rigid shift to apply to the atomic coordinates, given in the same format and units as these. Notice that the atomic positions (shifted or not) need not be within the cell formed by **LatticeVectors**, since periodic boundary conditions are always assumed.

This defaults to the origo:

```
0.0   0.0   0.0
```

%block AtomicCoordinatesAndAtomicSpecies ⟨**None**⟩ *(block)*

Block specifying the position and species of each atom. One line per atom, the reading is done this way:

```
From ia = 1 to natoms
     read: xa(ix,ia), isa(ia)
```

where `xa(ix,ia)` is the `ix` coordinate of atom `iai` in the format (units) specified by **AtomicCoordinatesFormat**, and `isa(ia)` is the species index of atom `ia`.

NOTE: this block *must* be present in the fdf file. If **NumberOfAtoms** is not specified, **NumberOfAtoms** will be defaulted to the number of atoms in this block.

NOTE: Zmatrix has precedence if specified.

6.4.2 Z-matrix format and constraints

The advantage of the traditional format is that it is much easier to set up a system. However, when working on systems with constraints, there are only a limited number of (very simple) constraints that may be expressed within this format, and recompilation is needed for each new constraint.

For any more involved set of constraints, a full **Zmatrix** formulation should be used - this offers much more control, and may be specified fully at run time (thus not requiring recompilation) - but it is more work to generate the input files for this form.

%block Zmatrix ⟨None⟩ *(block)*

This block provides a means for inputting the system geometry using a Z-matrix format, as well as controlling the optimization variables. This is particularly useful when working with molecular systems or restricted optimizations (such as locating transition states or rigid unit movements). The format also allows for hybrid use of Z-matrices and Cartesian or fractional blocks, as is convenient for the study of a molecule on a surface. As is always the case for a Z-matrix, the responsibility falls to the user to chose a sensible relationship between the variables to avoid triads of atoms that become linear.

Below is an example of a Z-matrix input for a water molecule:

```
%block Zmatrix
molecule fractional
  1 0 0 0   0.0 0.0 0.0 0 0 0
  2 1 0 0   HO1 90.0 37.743919 1 0 0
  2 1 2 0   HO2 HOH 90.0 1 1 0
variables
    HO1 0.956997
    HO2 0.956997
    HOH 104.4
%endblock Zmatrix
```

The sections that can be used within the Zmatrix block are as follows:

Firstly, all atomic positions must be specified within either a "`molecule`" block or a "`cartesian`" block. Any atoms subject to constraints more complicated than "do not change this coordinate of this atom" must be specified within a "`molecule`" block.

molecule There must be one of these blocks for each independent set of constrained atoms within the simulation.

This specifies the atoms that make up each molecule and their geometry. In addition, an option of "`fractional`" or "`scaled`" may be passed, which indicates that distances are specified in scaled or fractional units. In the absence of such an option, the distance units are taken to be the value of "`ZM.UnitsLength`".

A line is needed for each atom in the molecule; the format of each line should be:

```
Nspecies i j k r a t ifr ifa ift
```

Here the values `Nspecies`, `i`, `j`, `k`, `ifr`, `ifa`, and `ift` are integers and `r`, `a`, and `t` are double precision reals.

For most atoms, `Nspecies` is the species number of the atom, `r` is distance to atom number `i`, `a` is the angle made by the present atom with atoms `j` and `i`, while `t` is the torsional angle made by the present atom with atoms `k`, `j`, and `i`. The values `ifr`, `ifa` and `ift` are integer flags that indicate whether `r`, `a`, and `t`, respectively, should be varied; 0 for fixed, 1 for varying.

The first three atoms in a molecule are a special case. Because there are insufficient atoms defined to specify a distance/angle/torsion, the values are set differently. For atom 1, `r`, `a`, and `t`, are the Cartesian coordinates of the atom. For the second atom, `r`, `a`, and `t` are the coordinates in spherical form of the second atom relative to the first: first the radius, then the

polar angle (angle between the z-axis and the displacement vector) and then the azimuthal angle (angle between the x-axis and the projection of the displacement vector on the x-y plane). Finally, for the third atom, the numbers take their normal form, but the torsional angle is defined relative to a notional atom 1 unit in the z-direction above the atom `j`.

Secondly. blocks of atoms all of which are subject to the simplest of constraints may be specified in one of the following three ways, according to the units used to specify their coordinates:

cartesian This section specifies a block of atoms whose coordinates are to be specified in Cartesian coordinates. Again, an option of "`fractional`" or "`scaled`" may be added, to specify the units used; and again, in their absence, the value of "`ZM.UnitsLength`" is taken.

The format of each atom in the block will look like:

```
Nspecies x y z ix iy iz
```

Here `Nspecies`, `ix`, `iy`, and `iz` are integers and `x`, `y`, `z` are reals. `Nspecies` is the species number of the atom being specified, while `x`, `y`, and `z` are the Cartesian coordinates of the atom in whichever units are being used. The values `ix`, `iy` and `iz` are integer flags that indicate whether the `x`, `y`, and `z` coordinates, respectively, should be varied or not. A value of 0 implies that the coordinate is fixed, while 1 implies that it should be varied. **NOTE**: When performing "variable cell" optimization while using a Zmatrix format for input, the algorithm will not work if some of the coordinates of an atom in a `cartesian` block are variables and others are not (i.e., `ix iy iz` above must all be 0 or 1). This will be fixed in future versions of the program.

A Zmatrix block may also contain the following, additional, sections, which are designed to make it easier to read.

constants Instead of specifying a numerical value, it is possible to specify a symbol within the above geometry definitions. This section allows the user to define the value of the symbol as a constant. The format is just a symbol followed by the value:

```
HOH 104.4
```

variables Instead of specifying a numerical value, it is possible to specify a symbol within the above geometry definitions. This section allows the user to define the value of the symbol as a variable. The format is just a symbol followed by the value:

```
HO1 0.956997
```

Finally, constraints must be specified in a **constraints** block.

constraint This sub-section allows the user to create constraints between symbols used in a Z-matrix:

```
constraint
  var1 var2 A B
```

Here **var1** and **var2** are text symbols for two quantities in the Z-matrix definition, and $A and $B are real numbers. The variables are related by $\mathbf{var1} = A * \mathbf{var2} + B$.

An example of a Z-matrix input for a benzene molecule over a metal surface is:

```
%block Zmatrix
  molecule
   2 0 0 0 xm1 ym1 zm1 0 0 0
   2 1 0 0 CC 90.0 60.0 0 0 0
```

```
  2 2 1 0 CC CCC 90.0 0 0 0
  2 3 2 1 CC CCC 0.0 0 0 0
  2 4 3 2 CC CCC 0.0 0 0 0
  2 5 4 3 CC CCC 0.0 0 0 0
  1 1 2 3 CH CCH 180.0 0 0 0
  1 2 1 7 CH CCH 0.0 0 0 0
  1 3 2 8 CH CCH 0.0 0 0 0
  1 4 3 9 CH CCH 0.0 0 0 0
  1 5 4 10 CH CCH 0.0 0 0 0
  1 6 5 11 CH CCH 0.0 0 0 0
 fractional
  3 0.000000 0.000000 0.000000 0 0 0
  3 0.333333 0.000000 0.000000 0 0 0
  3 0.666666 0.000000 0.000000 0 0 0
  3 0.000000 0.500000 0.000000 0 0 0
  3 0.333333 0.500000 0.000000 0 0 0
  3 0.666666 0.500000 0.000000 0 0 0
  3 0.166667 0.250000 0.050000 0 0 0
  3 0.500000 0.250000 0.050000 0 0 0
  3 0.833333 0.250000 0.050000 0 0 0
  3 0.166667 0.750000 0.050000 0 0 0
  3 0.500000 0.750000 0.050000 0 0 0
  3 0.833333 0.750000 0.050000 0 0 0
  3 0.000000 0.000000 0.100000 0 0 0
  3 0.333333 0.000000 0.100000 0 0 0
  3 0.666666 0.000000 0.100000 0 0 0
  3 0.000000 0.500000 0.100000 0 0 0
  3 0.333333 0.500000 0.100000 0 0 0
  3 0.666666 0.500000 0.100000 0 0 0
  3 0.166667 0.250000 0.150000 0 0 0
  3 0.500000 0.250000 0.150000 0 0 0
  3 0.833333 0.250000 0.150000 0 0 0
  3 0.166667 0.750000 0.150000 0 0 0
  3 0.500000 0.750000 0.150000 0 0 0
  3 0.833333 0.750000 0.150000 0 0 0
 constants
  ym1 3.68
 variables
  zm1 6.9032294
  CC 1.417
  CH 1.112
  CCH 120.0
  CCC 120.0
 constraints
  xm1 CC -1.0 3.903229
%endblock Zmatrix
```

Here the species 1, 2 and 3 represent H, C, and the metal of the surface, respectively.

(Note: the above example shows the usefulness of symbolic names for the relevant coordinates, in particular for those which are allowed to vary. The current output options for Zmatrix information work best when this approach is taken. By using a "fixed" symbolic Zmatrix block and specifying the actual coordinates in a "variables" section, one can monitor the progress of the optimization and easily reconstruct the coordinates of intermediate steps in the original

format.)

ZM.UnitsLength **Bohr** *(string)*

Parameter that specifies the units of length used during Z-matrix input.

Specify **Bohr** or **Ang** for the corresponding unit of length.

ZM.UnitsAngle **rad** *(string)*

Parameter that specifies the units of angles used during Z-matrix input.

Specify **rad** or **deg** for the corresponding unit of angle.

6.4.3 Output of structural information

SIESTA is able to generate several kinds of files containing structural information (maybe too many).

- `SystemLabel.STRUCT_OUT`:Siesta always produces a `.STRUCT_OUT` file with cell vectors in Å and atomic positions in fractional coordinates. This file, renamed to `.STRUCT_IN` can be used for crystal-structure input. Note that the geometry reported is the last one for which forces and stresses were computed. See **UseStructFile**

- `SystemLabel.STRUCT_NEXT_ITER`:This file is always written, in the same format as `.STRUCT_OUT` file. The only difference is that it contains the structural information *after* it has been updated by the relaxation or the molecular-dynamics algorithms, and thus it could be used as input (renamed as `.STRUCT_IN`) for a continuation run, in the same way as the `.XV` file.

  See **UseStructFile**

- `SystemLabel.XV`:The coordinates are always written in the `.XV` file, and overriden at every step.

- `OUT.UCELL.ZMATRIX`:This file is produced if the Zmatrix format is being used for input. (Please note that **SystemLabel** is not used as a prefix.) It contains the structural information in fdf form, with blocks for unit-cell vectors and for Zmatrix coordinates. The Zmatrix block is in a "canonical" form with the following characteristics:

  ```
  1. No symbolic variables or constants are used.
  2. The position coordinates of the first atom in each molecule
     are absolute Cartesian coordinates.
  3. Any coordinates in ‘‘cartesian’’ blocks are also absolute Cartesians.
  4. There is no provision for output of constraints.
  5. The units used are those initially specified by the user, and are
     noted also in fdf form.
  ```

  Note that the geometry reported is the last one for which forces and stresses were computed.

- `NEXT_ITER.UCELL.ZMATRIX`:A file with the same format as `OUT.UCELL.ZMATRIX` but with a possibly updated geometry.

- The coordinates can be also accumulated in the `SystemLabel.MD` or `SystemLabel.MDX` files depending on **WriteMDHistory**.
- Additionally, several optional formats are supported:

  WriteCoorXmol **false** *(logical)*

  If **true** it originates the writing of an extra file named `SystemLabel.xyz` containing the final atomic coordinates in a format directly readable by XMol.[4] Coordinates come out in Ångström independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. There is a present Java implementation of XMol called JMol.

  WriteCoorCerius **false** *(logical)*

  If **true**it originates the writing of an extra file named `SystemLabel.xtl` containing the final atomic coordinates in a format directly readable by Cerius.[5] Coordinates come out in **Fractional** format (the same as **ScaledByLatticeVectors**) independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. If negative coordinates are to be avoided, it has to be done from the start by shifting all the coordinates rigidly to have them positive, by using **AtomicCoordinatesOrigin**. See the `Sies2arc` utility in the `Util/` directory for generating `..arc` files for CERIUS animation.

  WriteMDXmol **false** *(logical)*

  If **true** it causes the writing of an extra file named `SystemLabel.ANI` containing all the atomic coordinates of the simulation in a format directly readable by XMol for animation. Coordinates come out in Ångström independently of what is specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. This file is accumulative even for different runs.

  There is an alternative for animation by generating a `.arc` file for CERIUS. It is through the Sies2arc postprocessing utility in the Util/ directory, and it requires the coordinates to be accumulated in the output file, i.e., **WriteCoorStep true**.

6.4.4 Input of structural information from external files

The structural information can be also read from external files. Note that **ChemicalSpeciesLabel** is mandatory in the fdf file.

MD.UseSaveXV **false** *(logical)*

Logical variable which instructs SIESTA to read the atomic positions and velocities stored in file `SystemLabel.XV` by a previous run.

If the file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**, but can be implicitly set by it.

UseStructFile **false** *(logical)*

Controls whether the structural information is read from an external file of name `SystemLabel.STRUCT_IN`. If **true**, all other structural information in the fdf file will be ignored.

[4] XMol is under © copyright of Research Equipment Inc., dba Minnesota Supercomputer Center Inc.
[5] Cerius is under © copyright of Molecular Simulations Inc.

The format of the file is implied by the following code:

```
read(*,*) ((cell(ixyz,ivec),ixyz=1,3),ivec=1,3)  ! Cell vectors, in Angstroms
read(*,*) na
do ia = 1,na
   read(iu,*) isa(ia), dummy, xfrac(1:3,ia)  ! Species number
                                             ! Dummy numerical column
                                             ! Fractional coordinates
enddo
```

Warning: Note that the resulting geometry could be clobbered if an `.XV` file is read after this file. It is up to the user to remove any `.XV` files.

MD.UseSaveZM **false** *(logical)*

Instructs to read the Zmatrix information stored in file `.ZM` by a previous run.

If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**, but can be implicitly set by it.

Warning: Note that the resulting geometry could be clobbered if an `.XV` file is read after this file. It is up to the user to remove any `.XV` files.

6.4.5 Input from a FIFO file

See the "Forces" option in **MD.TypeOfRun**. Note that **ChemicalSpeciesLabel** is still mandatory in the fdf file.

6.4.6 Precedence issues in structural input

- If the "Forces" option is active, it takes precedence over everything (it will overwrite all other input with the information it gets from the FIFO file).
- If **MD.UseSaveXV** is active, it takes precedence over the options below.
- If **UseStructFile** (or **MD.UseStructFile**) is active, it takes precedence over the options below.
- For atomic coordinates, the traditional and Zmatrix formats in the fdf file are mutually exclusive. If **MD.UseSaveZM** is active, the contents of the ZM file, if found, take precedence over the Zmatrix information in the fdf file.

6.4.7 Interatomic distances

WarningMinimumAtomicDistance 1 Bohr *(length)*

Fixes a threshold interatomic distance below which a warning message is printed.

MaxBondDistance 6 Bohr *(length)*

SIESTA prints the interatomic distances, up to a range of **MaxBondDistance**, to file `SystemLabel.BONDS` upon first reading the structural information, and to file `SystemLabel.BONDS_FINAL` after the last geometry iteration. The reference atoms are all the

atoms in the unit cell. The routine now prints the real location of the neighbor atoms in space, and not, as in earlier versions, the location of the equivalent representative in the unit cell.

6.5 k-point sampling

These are options for the k-point grid used in the SCF cycle. For other specialized grids, see the Macroscopic Polarization and Density of States sections.

kgrid.Cutoff 0. Bohr *(length)*

Parameter which determines the fineness of the k-grid used for Brillouin zone sampling. It is half the length of the smallest lattice vector of the supercell required to obtain the same sampling precision with a single k point. Ref: Moreno and Soler, PRB 45, 13891 (1992).

Use: If it is zero, only the gamma point is used. The resulting k-grid is chosen in an optimal way, according to the method of Moreno and Soler (using an effective supercell which is as spherical as possible, thus minimizing the number of k-points for a given precision). The grid is displaced for even numbers of effective mesh divisions. This parameter is not used if **kgrid.MonkhorstPack** is specified. If the unit cell changes during the calculation (for example, in a cell-optimization run, the k-point grid will change accordingly (see **ChangeKgridInMD** for the case of variable-cell molecular-dynamics runs, such as Parrinello-Rahman). This is analogous to the changes in the real-space grid, whose fineness is specified by an energy cutoff. If sudden changes in the number of k-points are not desired, then the Monkhorst-Pack data block should be used instead. In this case there will be an implicit change in the quality of the sampling as the cell changes. Both methods should be equivalent for a well-converged sampling.

%block kgrid.MonkhorstPack Γ**-point** *(block)*

Real-space supercell, whose reciprocal unit cell is that of the k-sampling grid, and grid displacement for each grid coordinate. Specified as an integer matrix and a real vector:

```
%block kgrid.MonkhorstPack
   Mk(1,1)  Mk(2,1)  Mk(3,1)   dk(1)
   Mk(1,2)  Mk(2,2)  Mk(3,2)   dk(2)
   Mk(1,3)  Mk(2,3)  Mk(3,3)   dk(3)
%endblock
```

where `Mk(j,i)` are integers and `dk(i)` are usually either 0.0 or 0.5 (the program will warn the user if the displacements chosen are not optimal). The k-grid supercell is defined from `Mk` as in block **SuperCell** above, i.e.: $KgridSuperCell(ix, i) = \sum_j CELL(ix, j) * Mk(j, i)$. Note again that the matrix indexes are inverted: each input line gives the decomposition of a supercell vector in terms of the unit cell vectors.

Use: Used only if **SolutionMethod diagon**. The k-grid supercell is compatible and unrelated (except for the default value, see below) with the **SuperCell** specifier. Both supercells are given in terms of the CELL specified by the **LatticeVectors** block. If `Mk` is the identity matrix and `dk` is zero, only the Γ point of the **unit** cell is used. Overrides **kgrid.Cutoff**

ChangeKgridInMD **false** *(logical)*

If **true**, the k-point grid is recomputed at every iteration during MD runs that potentially change the unit cell: Parrinello-Rahman, Nose-Parrinello-Rahman, and Anneal. Regardless of the setting of this flag, the k-point grid is always updated at every iteration of a variable-cell

optimization and after each step in a "siesta-as-server" run.

It is defaulted to **false** for historical reasons. The rationale was to avoid sudden jumps in some properties when the sampling changes, but if the calculation is well-converged there should be no problems if the update is enabled.

TimeReversalSymmetryForKpoints **true** *(logical)*

If **true**, the k-points in the BZ generated by the methods above are paired as $(k,-k)$ and only one member of the pair is retained. This symmetry is valid in the absence of external magnetic fields or spin-orbit interaction.

Note this defaults to **false** if **Spin** is **non-collinear**, **spin-orbit** or **Spin.Spiral** is used.

6.5.1 Output of k-point information

The coordinates of the $\vec{k}$ points used in the sampling are always stored in the file `SystemLabel.KP`.

WriteKpoints **false** *(logical)*

If **true** it writes the coordinates of the $\vec{k}$ vectors used in the grid for k-sampling, into the main output file.

Default depends on **LongOutput**.

6.6 Exchange-correlation functionals

XC.Functional **LDA** *(string)*

Exchange-correlation functional type. May be **LDA** (local density approximation, equivalent to **LSD**), **GGA** (Generalized Gradient Approximation), or **VDW** (van der Waals).

XC.Authors **PZ** *(string)*

Particular parametrization of the exchange-correlation functional. Options are:

- **CA** (equivalent to **PZ**): (Spin) local density approximation (LDA/LSD). Quantum Monte Carlo calculation of the homogeneous electron gas by D. M. Ceperley and B. J. Alder, Phys. Rev. Lett. **45**,566 (1980), as parametrized by J. P. Perdew and A. Zunger, Phys. Rev B **23**, 5075 (1981)
- **PW92**: LDA/LSD, as parametrized by J. P. Perdew and Y. Wang, Phys. Rev B, **45**, 13244 (1992)
- **PW91**: Generalized gradients approximation (GGA) of Perdew and Wang. Ref: P&W, J. Chem. Phys., **100**, 1290 (1994)
- **PBE**: GGA of J. P. Perdew, K. Burke and M. Ernzerhof, Phys. Rev. Lett. **77**, 3865 (1996)
- **revPBE**: Modified GGA-PBE functional of Y. Zhang and W. Yang, Phys. Rev. Lett. **80**, 890 (1998)
- **RPBE**: Modified GGA-PBE functional of B. Hammer, L. B. Hansen and J. K. Norskov Phys. Rev. B **59**, 7413 (1999)
- **WC**: Modified GGA-PBE functional of Z. Wu and R. E. Cohen, Phys. Rev. B **73**, 235116 (2006)

- **AM05**: Modified GGA-PBE functional of R. Armiento and A. E. Mattsson, Phys. Rev. B **72**, 085108 (2005)
- **PBEsol**: Modified GGA-PBE functional of J. P. Perdew et al, Phys. Rev. Lett. **100**, 136406 (2008)
- **PBEJsJrLO**: GGA-PBE functional with parameters β, μ, and κ fixed by the jellium surface (Js), jellium response (Jr), and Lieb-Oxford bound (LO) criteria, respectively, as described by L. S. Pedroza, A. J. R. da Silva, and K. Capelle, Phys. Rev. B **79**, 201106(R) (2009), and by M. M. Odashima, K. Capelle, and S. B. Trickey, J. Chem. Theory Comput. **5**, 798 (2009)
- **PBEJsJrHEG**: Same as PBEJsJrLO, with parameter κ fixed by the Lieb-Oxford bound for the low density limit of the homogeneous electron gas (HEG)
- **PBEGcGxLO**: Same as PBEJsJrLO, with parameters β and μ fixed by the gradient expansion of correlation (Gc) and exchange (Gx), respectively
- **PBEGcGxHEG**: Same as previous ones, with parameters β, μ, and κ fixed by the Gc, Gx, and HEG criteria, respectively.
- **BLYP** (equivalent to **LYP**): GGA with Becke exchange (A. D. Becke, Phys. Rev. A **38**, 3098 (1988)) and Lee-Yang-Parr correlation (C. Lee, W. Yang, R. G. Parr, Phys. Rev. B **37**, 785 (1988)), as modified by B. Miehlich, A. Savin, H. Stoll, and H. Preuss, Chem. Phys. Lett. **157**, 200 (1989). See also B. G. Johnson, P. M. W. Gill and J. A. Pople, J. Chem. Phys. **98**, 5612 (1993). (Some errors were detected in this last paper, so not all of their expressions correspond exactly to those implemented in SIESTA)
- **DRSLL** (equivalent to **DF1**): van der Waals density functional (vdW-DF) of M. Dion, H. Rydberg, E. Schröder, D. C. Langreth, and B. I. Lundqvist, Phys. Rev. Lett. **92**, 246401 (2004), with the efficient implementation of G. Román-Pérez and J. M. Soler, Phys. Rev. Lett. **103**, 096102 (2009)
- **LMKLL** (equivalent to **DF2**): vdW-DF functional of Dion *et al* (same as DRSLL) reparametrized by K. Lee, E. Murray, L. Kong, B. I. Lundqvist and D. C. Langreth, Phys. Rev. B **82**, 081101 (2010)
- **KBM**: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by J. Klimes, D. R. Bowler, and A. Michaelides, J. Phys.: Condens. Matter **22**, 022201 (2010) (optB88-vdW version)
- **C09**: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by V. R. Cooper, Phys. Rev. B **81**, 161104 (2010)
- **BH**: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by K. Berland and P. Hyldgaard, Phys. Rev. B 89, 035412 (2014)
- **VV**: vdW-DF functional of O. A. Vydrov and T. Van Voorhis, J. Chem. Phys. **133**, 244103 (2010)

%block XC.Hybrid ⟨**None**⟩ *(block)*

This data block allows the user to create a "cocktail" functional by mixing the desired amounts of exchange and correlation from each of the functionals described under XC.authors. Note that these "mixed" functionals do *not* have the exact Hartree-Fock exchange which is a key ingredient of the true "hybrid" functionals. The use of the word "hybrid" in the label is unfortunate in this regard, and might be deprecated in a future version.

The first line of the block must contain the number of functionals to be mixed. On the subsequent lines the values of XC.functl and XC.authors must be given and then the weights for the exchange and correlation, in that order. If only one number is given then the same weight is applied to both exchange and correlation.

The following is an example in which a 75:25 mixture of Ceperley-Alder and PBE correlation is made, with an equal split of the exchange energy:

```
%block XC.hybrid
   2
   LDA CA  0.5 0.75
   GGA PBE 0.5 0.25
%endblock XC.hybrid
```

6.7 Spin polarization

Spin **non-polarized** *(string)*

Choose the spin-components in the simulation.

NOTE: this flag has precedence over **SpinOrbit**, **NonCollinearSpin** and **SpinPolarized** while these older flags may still be used.

non-polarized Perform a calculation with spin-degeneracy (only one component).

polarized Perform a calculation with collinear spin (two spin components).

non-collinear Perform a calculation with non-collinear spin (4 spin components), up-down and angles.

Refs: T. Oda et al, PRL, **80**, 3622 (1998); V. M. García-Suárez et al, Eur. Phys. Jour. B **40**, 371 (2004); V. M. García-Suárez et al, Journal of Phys: Cond. Matt **16**, 5453 (2004).

spin-orbit Perform a calculation with spin-orbit coupling. This requires the pseudopotentials to be relativistic.

See Sect. 6.8.

SIESTA can read a `.DM` with different spin structure by adapting the information to the currently selected spin multiplicity, averaging or splitting the spin components equally, as needed. This may be used to greatly increase convergence.

Certain options may not be used together with specific parallelization routines. For instance only a spin-polarized calculation may use the **Diag.ParallelOverK** option.

Spin.Fix **false** *(logical)*

If **true**, the calculation is done with a fixed value of the spin of the system, defined by variable **Spin.Total**. This option can only be used for collinear spin polarized calculations.

Spin.Total 0 *(real)*

Value of the imposed total spin polarization of the system (in units of the electron spin, 1/2). It is only used if **Spin.Fix true**.

SingleExcitation **false** *(logical)*

If **true**, SIESTA calculates a very rough approximation to the lowest excited state by swapping the populations of the HOMO and the LUMO. If there is no spin polarisation, it is half swap only. It is done for the first spin component (up) and first k vector.

6.8 Spin–Orbit coupling

SIESTA includes the posibility to perform fully relativistic calculations by means of the inclusion in the total Hamiltonian not only the Darwin and velocity correction terms (Scalar–Relativistic calculations), but also the spin-orbit (SO) contribution. The implementation is based on the on-site SO approximation, where only the intra-SO contribution of each atom is taken into account. See **Spin** on how to turn on the spin-orbit coupling.

The current implementation in SIESTA has been implemented by Dr. Ramón Cuadrado based on the original on-site SO formalism and implementation developed by Prof. Jaime Ferrer, *et al* (L Fernández–Seivane, M Oliveira, S Sanvito, and J Ferrer, Journal of Physics: Condensed Matter, 2006 vol. 18 pp. 7999; L Fernández–Seivane and Jaime Ferrer, Phys. Rev. Lett. 99, 2007, 183401).

The inclusion of the SO term in the Hamiltonian (and in the Density Matrix) will involve the increase of non-zero elements in their off-diagonal parts, i.e., for some $\mu\nu$ orbitals, $H_{\mu\nu}^{\sigma\sigma'}(DM_{\mu\nu}^{\sigma\sigma'})$ $[\sigma, \sigma'=\uparrow,\downarrow]$ will be $\neq 0$. This is mainly due to the fact that the $\mathbf{L}\cdot\mathbf{S}$ operator will promote the mixing between different spin-up/down components. The terms responsible of this matrices expansion are the exchange-correlation potential and the SO. The remaining terms such as the kinetic energy or Hartree contribution do not depend of the spin orientations and hence will be only added to the total Hamiltonian (and DM) to their diagonal parts.

The current SO formalism enables the possibility of several types of calculations:

- Selfconsistent calculations for gamma point as well as for bulks (Not yet implemented for optimizations).
- Magnetic Anisotropy Energy (MAE) can be easily calculated. From first principles calculations, MAE is obtained after subtract the total selfconsistent energy in two different orientations, usually the total energy associated with easy axis from the hard axis. In SIESTA it is possible to perform several self-consistent calculations for different magnetization orientations using the specific block **DM.InitSpin** in the fdf file. In doing so one will be able to include the initial orientation angles of the magnetization for each atom, as well as an initial value of their net magnetic moments.
- By means of Mulliken analysis, after the self-consistent procedure, local spin and orbital moments can be calculated by means of the flag **WriteOrbMom**.

Note: Due to the small SO energy value contribution to the total energy, the level of precision requiered to perform a proper fully relativistic calculation during the selfconsistent process is quite demanding. The following values must be carefully converged and checked for each specific system to assure that the results are accurate enough: **SCF.H.Tolerance** during the selfconsistency (typically $<10^{-5}$eV), **ElectronicTemperature**, **k**-point sampling and high values of **MeshCutoff** (specifically for extended solids). In general, one can say that a good calculation will have high number of k–points, low **ElectronicTemperature**, extremely small **SCF.H.Tolerance** and high values of **MeshCutoff**. We encourage the user to test carefully these options for each system. An additional point to take into account when the spin–orbit contribution is included is the mixing scheme employed. You are encouraged to use **SCF.Mix hamiltonian** instead of the density matrix, due to the fact that the convergence speed increases considerably for the first case. In addition, the pseudopotentials have to be well generated and tested for each specific system and they have to be generated in their fully relativistic form and use the non-linear core corrections.

Spin.OrbitStrength **1.0** *(real)*

It allows to vary the strength of the spin–orbit interaction from zero to any positive value, including the physical value. This flag is only active when SpinOrbit is set to **true**.

WriteOrbMom **false** *(logical)*

If **true**, a table is provided in the main output file, which includes an estimation of the vector spin and orbital magnetic moments, in units of the Bohr magneton, projected onto each orbital and also onto each atom. The estimation for the orbital moments is based on a two-center approximation, and makes use of the Mulliken population analysis.

6.9 The self-consistent-field loop

IMPORTANT NOTE: Convergence of the Kohn-Sham energy and forces

In versions prior to 4.0 of the program, the Kohn-Sham energy was computed using the "in" DM. The typical DM used as input for the calculation of H was not directly computed from a set of wave-functions (it was either the product of mixing or of the initialization from atomic values). In this case, the "kinetic energy" term in the total energy computed in the way stated in the Siesta paper had an error which decreased with the approach to self-consistency, but was non-zero. The net result was that the Kohn-Sham energy converged more slowly than the "Harris" energy (which is correctly computed).

When mixing H (see below under "Mixing Options"), the KS energy is in effect computed from DM(out), so this error vanishes.

As a related issue, the forces and stress computed after SCF convergence were calculated using the DM coming out of the cycle, which by default was the product of a final mixing. This also introduced errors which grew with the degree of non-selfconsistency.

The current version introduces several changes:

- When mixing the DM, the Kohn-Sham energy may be corrected to make it variational. This involves an extra call to `dhscf` (although with neither forces nor matrix elements being calculated, i.e. only calls to `rhoofd`, `poison`, and `cellxc`), and is turned on by the option **SCF.Want.Variational.EKS**.

- The program now prints a new column labeled "dHmax" for the self-consistent cycle. The value represents the maximum absolute value of the changes in the entries of H, but its actual meaning depends on whether DM or H mixing is in effect: if mixing the DM, dHmax refers to the change in H(in) with respect to the previous step; if mixing H, dHmax refers to H(out)-H(in) in the previous(?) step.

- When achieving convergence, the loop might be exited without a further mixing of the DM, thus preserving DM(out) for further processing (including the calculation of forces and the analysis of the electronic structure) (see the **SCF.MixAfterConvergence** option).

- It remains to be seen whether the forces, being computed "right" on the basis of DM(out), exhibit somehow better convergence as a function of the scf step. In order to gain some more data and heuristics on this we have implemented a force-monitoring option, activated by setting to **true** the variable **SCF.MonitorForces**. The program will then print the maximum

absolute value of the change in forces from one step to the next. Other statistics could be implemented.

- While the (mixed) DM is saved at every SCF step, as was standard practice, the final DM(out) overwrites the .DM file at the end of the SCF cycle. Thus it is still possible to use a "mixed" DM for restarting an interrupted loop, but a "good" DM will be used for any other post-processing.

6.9.1 Harris functional and basic options

Harris.Functional **false** *(logical)*

Logical variable to choose between self-consistent Kohn-Sham functional or non self-consistent Harris functional to calculate energies and forces.

- **false**: Fully self-consistent Kohn-Sham functional.
- **true**: Non self consistent Harris functional. Cheap but pretty crude for some systems. The forces are computed within the Harris functional in the first SCF step. Only implemented for LDA in the Perdew-Zunger parametrization. It really only applies to starting densities which are superpositions of atomic charge densities.

  When this option is choosen, the values of **DM.UseSaveDM**, **SCF.MustConverge** and **SCF.Mix.First** are automatically set **false**and **MaxSCFIterations** is set to 1, no matter whatever other specification are in the INPUT file.

MinSCFIterations **0** *(integer)*

Minimum number of SCF iterations per time step. In MD simulations this can with benefit be set to 3.

MaxSCFIterations **50** *(integer)*

Maximum number of SCF iterations per time step.

SCF.MustConverge **false** *(logical)*

Defines the behaviour if convergence is not reached in the maximum number of SCF iterations. The default is to update the forces, perform an MD or geometry optimisation step and carry on. When set to true the calculation will stop on the first SCF convergence failure.

6.9.2 Mixing options

Whether a calculation reaches self-consistency in a moderate number of steps depends strongly on the mixing parameters used. The available mixing options should be carefully tested for a given calculation type. This search for optimal parameters can repay itself handsomely by potentially saving many self-consistency steps in production runs.

SCF.Mix **Hamiltonian|density|charge** *(string)*

Control what physical quantity to mix in the self-consistent cycle.

The default is mixing the Hamiltonian, which may typically perform better than density matrix mixing.

Hamiltonian Mix the Hamiltonian matrix (default).

density Mix the density matrix.

charge Mix the real-space charge density. Note this is an experimental feature.

NOTE: Real-space charge density does not follow the regular options that adhere to density-matrix or Hamiltonian mixing.

SCF.Mix.Spin **all|spinor|sum|sum+diff** *(string)*

Controls how the mixing is performed when carrying out spin-polarized calculations.

all Use all spin-components in the mixing

spinor Estimate mixing coefficients using the spinor components

sum Estimate mixing coefficients using the sum of the spinor components

sum+diff Estimate mixing coefficients using the sum *and* the difference between the spinor components

NOTE: this option only influences density-matrix ($\boldsymbol{\rho}$) or Hamiltonian ($\mathbf{H}$) mixing when using anything but the **linear** mixing scheme. And it does not influence not charge (ρ) mixing.

SCF.Mix.First **true** *(logical)*

Whether the first SCF should be mixed or it uses the output as input in the next SCF step. It is generally advised to set this to true, at least when restarting calculations.

In the following the density matrix ($\boldsymbol{\rho}$) will be used in the equations, while for Hamiltonian mixing, $\boldsymbol{\rho}$, should be replaced by the Hamiltonian matrix. Also we define $\mathrm{R}[i] = \boldsymbol{\rho}^i_{\mathrm{out}} - \boldsymbol{\rho}^i_{\mathrm{in}}$ and $\Delta\mathrm{R}[i] = \mathrm{R}[i] - \mathrm{R}[i-1]$.

SCF.Mixer.Method **Pulay|Broyden|Linear** *(string)*

Choose the mixing algorithm between different methods. Each method may have different variants, see **SCF.Mixer.Variant**.

Linear A simple linear extrapolation of the input matrix as

$$\boldsymbol{\rho}^{n+1}_{\mathrm{in}} = \boldsymbol{\rho}^{n}_{\mathrm{in}} + w\,\mathrm{R}[n]. \tag{3}$$

Pulay Using the Pulay mixing method corresponds using the [**?**] variant. It relies on the previous N steps and uses those for estimating an optimal input $\boldsymbol{\rho}^{n+1}_{\mathrm{in}}$ for the following iteration. The equation can be written as

$$\boldsymbol{\rho}^{n+1}_{\mathrm{in}} = \boldsymbol{\rho}^{n}_{\mathrm{in}} + G\,\mathrm{R}[n] + \sum_{i=n-N+1}^{N-1} \alpha_i(\mathrm{R}[i] + G\,\Delta\mathrm{R}[i]), \tag{4}$$

where G is the damping factor of the Pulay mixing (also known as the mixing weight). The values α_i are calculated using this formula

$$\alpha_i = -\sum_{j=1}^{N-1} \mathbf{A}^{-1}_{ji} \langle \Delta\mathrm{R}[j] |\, \mathrm{R}[N] \rangle, \tag{5}$$

with $\mathbf{A}_{ji} = \langle \Delta\mathrm{R}[j] |\, \Delta\mathrm{R}[i] \rangle$.

In SIESTA G is a constant, and not a matrix.

NOTE: Pulay mixing is a special case of Broyden mixing, see the Broyden method.

Broyden The Broyden mixing is mixing method relying on the previous N steps in the history for calculating an optimum input $\boldsymbol{\rho}_{\mathrm{in}}^{n+1}$ for the following iteration. The equation can be written as

$$\boldsymbol{\rho}_{\mathrm{in}}^{n+1} = \boldsymbol{\rho}_{\mathrm{in}}^{n} + G\,\mathrm{R}[n] - \sum_{i=n-N+1}^{N-1} \sum_{j=n-N+1}^{N-1} w_i w_j c_j \beta_{ij} (\mathrm{R}[i] + G\,\Delta\mathrm{R}[i]), \tag{6}$$

where G is the damping factor (also known as the mixing weight). The values weights may be expressed by

$$w_i = 1 \quad \text{, for } i > 0 \tag{7}$$
$$c_i = \langle \Delta\mathrm{R}[i] |\, \mathrm{R}[n] \rangle, \tag{8}$$
$$\beta_{ij} = \left[\left(w_0^2 \mathbf{I} + \mathbf{A} \right)^{-1} \right]_{ij} \tag{9}$$
$$A_{ij} = w_i w_j \langle \Delta\mathrm{R}[i] |\, \Delta\mathrm{R}[j] \rangle. \tag{10}$$

It should be noted that w_i for $i > 0$ may be chosen arbitrarily. Comparing with the Pulay mixing scheme it is obvious that Broyden and Pulay are equivalent for a suitable set of parameters.

SCF.Mixer.Variant original *(string)*

Choose the variant of the mixing method.

Pulay This is implemented in two variants:

original|kresse The original[6] Pulay mixing scheme, as implemented in [**?**].

GR The "guaranteed-reduction" variant of Pulay, [**?**]. This variant has a special convergence path. It interchanges between linear and Pulay mixing thus using the exact gradient at each $\boldsymbol{\rho}_{\mathrm{in}}^{n}$. For relatively simple systems this may be advantageous to use. However, for complex systems it may be worse until it reaches a convergence basin.

To obtain the original guaranteed-reduction variant one should set **SCF.Mixer.<>.weight.linear** to 1.

SCF.Mixer.Weight 0.25 *(real)*

The mixing weight used to mix the quantity. In the linear mixing case this refers to

$$\boldsymbol{\rho}_{\mathrm{in}}^{n+1} = \boldsymbol{\rho}_{\mathrm{in}}^{n} + w\,\mathrm{R}[n]. \tag{11}$$

For details regarding the other methods please see **SCF.Mixer.Method**.

SCF.Mixer.History 2 *(integer)*

Number of previous SCF steps used in estimating the following input. Increasing this number, typically, increases stability and a number of around 6 or above may be advised.

SCF.Mixer.Kick 0 *(integer)*

After every N SCF steps a linear mix is inserted to *kick* the SCF cycle out of a possible local minimum.

The mixing weight for this linear kick is determined by **SCF.Mixer.Kick.Weight**.

[6] As such the "original" version is a variant it-self. But this is more stable in the far majority of cases.

SCF.Mixer.Kick.Weight ⟨**SCF.Mixer.Weight**⟩ *(real)*

The mixing weight for the linear kick (if used).

SCF.Mixer.Restart **0** *(integer)*

When using advanced mixers (Pulay/Broyden) the mixing scheme may periodically restart the history. This may greatly improve the convergence path as local constraints in the minimization process are periodically removed. This method has similarity to the method proposed in [**?**] and is a special case of the **SCF.Mixer.Kick** method.

Please see **SCF.Mixer.Restart.Save** which is advised to be set simultaneously.

SCF.Mixer.Restart.Save **1** *(integer)*

When restarting the history of saved SCF steps one may choose to save a subset of the latest history steps. When using **SCF.Mixer.Restart** it is encouraged to also save a couple of previous history steps.

SCF.Mixer.Linear.After **-1** *(integer)*

After reaching convergence one may run additional SCF cycles using a linear mixing scheme. If this has a value ≥ 0 SIESTA will perform linear mixing after it has converged using the regular mixing method (**SCF.Mixer.Method**).

The mixing weight for this linear mixing is controlled by **SCF.Mixer.Linear.After.Weight**.

SCF.Mixer.Linear.After.Weight ⟨**SCF.Mixer.Weight**⟩ *(real)*

After reaching convergence one may run additional SCF cycles using a linear mixing scheme. If this has a value ≥ 0 SIESTA will perform linear mixing after it has converged using the regular mixing method (**SCF.Mixer.Method**).

The mixing weight for this linear mixing is controlled by **SCF.Mixer.Linear.After.Weight**.

In conjunction with the above simple settings controlling the SCF cycle SIESTA employs a very configurable mixing scheme. In essence one may switch mixing methods, arbitrarily, during the SCF cycle via control commands. This can greatly speed up convergence.

%block SCF.Mixers ⟨**None**⟩ *(block)*

Each line in this block defines a separate mixer that is defined in a subsequent **SCF.Mixer.<>** block.

The first line is the initial mixer used.

See the following options for controlling individual mixing methods.

NOTE: If this block is defined you *must* define all mixing parameters individually.

%block SCF.Mixer.<> ⟨**None**⟩ *(block)*

This block controls the mixer named <>.

method Define the method for the mixer, see **SCF.Mixer.Method** for possible values.

variant Define the variant of the method, see **SCF.Mixer.Variant** for possible values.

weight|w Define the mixing weight for the mixing scheme, see **SCF.Mixer.Weight**.

history Define number of previous history steps used in the minimization process, see **SCF.Mixer.History**.

weight.linear|w.linear Define the linear mixing weight for the mixing scheme. This only has meaning for Pulay or Broyden mixing. It defines the initial linear mixing weight.

To obtain the original Pulay Guarenteed-Reduction variant one should set this to 1.

restart Define the periodic restart of the saved history, see **SCF.Mixer.Restart**.

restart.save Define number of latest history steps retained when restarting the history, see **SCF.Mixer.Restart.Save**.

iterations Define the maximum number of iterations this mixer should run before changing to another mixing method.

NOTE: this *must* be used in conjunction with the **next** setting.

next <> Specify the name of the next mixing scheme after having conducted **iterations** SCF cycles using this mixing method.

next.conv <> If SCF convergence is reached using this mixer, switch to the mixing scheme via <>. Then proceed with the SCF cycle.

next.p If the relative difference between the latest two residuals is below this quantity, the mixer will switch to the method given in **next**. Thus if

$$\frac{\langle \mathrm{R}[i]|\,\mathrm{R}[i]\rangle - \langle \mathrm{R}[i-1]|\,\mathrm{R}[i-1]\rangle}{\langle \mathrm{R}[i-1]|\,\mathrm{R}[i-1]\rangle} < \mathbf{next.p} \tag{12}$$

is fulfilled it will skip to the next mixer.

restart.p If the relative difference between the latest two residuals is below this quantity, the mixer will restart the history. Thus if

$$\frac{\langle \mathrm{R}[i]|\,\mathrm{R}[i]\rangle - \langle \mathrm{R}[i-1]|\,\mathrm{R}[i-1]\rangle}{\langle \mathrm{R}[i-1]|\,\mathrm{R}[i-1]\rangle} < \mathbf{restart.p} \tag{13}$$

is fulfilled it will reset the history.

The options covered now may be exemplified in these examples. If the input file contains:

```
SCF.Mixer.Method pulay
SCF.Mixer.Weight 0.05
SCF.Mixer.History 10
SCF.Mixer.Restart 25
SCF.Mixer.Restart.Save 4
SCF.Mixer.Linear.After 0
SCF.Mixer.Linear.After.Weight 0.1
```

This may be equivalently setup using the more advanced input blocks:

```
%block SCF.Mixers
  init
  final
%endblock

%block SCF.Mixer.init
   method pulay
```

```
   weight 0.05
   history 10
   restart 25
   restart.save 4
   next.conv final
%endblock

%block SCF.Mixer.final
   method linear
   weight 0.1
%endblock
```

This advanced setup may be used to change mixers during the SCF to change certain parameters of the mixing method, or fully change the method for mixing. For instance it may be advantageous to increase the mixing weight once a certain degree of self-consistency has been reached. In the following example we change the mixing method to a different scheme by increasing the weight and decreasing the history steps:

```
%block SCF.Mixers
  init
  final
%endblock

%block SCF.Mixer.init
   method pulay
   weight 0.05
   history 10
   next final
   # Switch when the relative residual goes below 5%
   next.p 0.05
%endblock

%block SCF.Mixer.final
   method pulay
   weight 0.1
   history 6
%endblock
```

In essence, very complicated schemes of convergence may be created using the block's input.

The following options refer to the global treatment of how/when mixing should be performed.

Compat.Pre-v4-DM-H **false** *(logical)*

This
controls the default values of **SCF.Mix.AfterConvergence**, **SCF.RecomputeHAfterScf** and **SCF.Mix.First**.

In versions prior to v4 the two former options where defaulted to **true** while the latter option was defaulted to **false**.

SCF.Mix.AfterConvergence **false** *(logical)*

Indicate whether mixing is done in the last SCF cycle (after convergence has been achieved) or not. Not mixing after convergence improves the quality of the final Kohn-Sham energy and of

the forces when mixing the DM.

NOTE: see **Compat.Pre-v4-DM-H**.

SCF.RecomputeHAfterSCF **false** *(logical)*

Indicate whether the Hamiltonian is updated after the scf cycle, while computing the final energy, forces, and stresses. Not recomputing H makes further analysis tasks (such as the computation of band structures) more consistent, as they will be able to use the same H used to generate the last density matrix.

NOTE: see **Compat.Pre-v4-DM-H**.

6.9.3 Mixing of the Charge Density

See **SCF.Mix** on how to enable charge density mixing. If charge density mixing is enabled the fourier components of the charge density are mixed, as done in some plane-wave codes. (See for example Kresse and Furthmüller, Comp. Mat. Sci. 6, 15-50 (1996), KF in what follows.)

The charge mixing is implemented roughly as follows:

- The charge density computed in dhscf is fourier-transformed and stored in a new module. This is done both for "$\rho(\mathbf{G})$(in)" and "$\rho(\mathbf{G})$(out)" (the "out" charge is computed during the extra call to dhscf for correction of the variational character of the Kohn-Sham energy)
- The "in" and "out" charges are mixed (see below), and the resulting "in" fourier components are used by dhscf in successive iterations to reconstruct the charge density.
- The new arrays needed and the processing of most new options is done in the new module m_rhog.F90. The fourier-transforms are carried out by code in rhofft.F.
- Following standard practice, two options for mixing are offered:
  - A simple Kerker mixing, with an optional Thomas-Fermi wavevector to damp the contributions for small G's. The overall mixing weight is the same as for other kinds of mixing, read from **DM.MixingWeight**.
  - A DIIS (Pulay) procedure that takes into account a sub-set of the G vectors (those within a smaller cutoff). Optionally, the scalar product used for the construction of the DIIS matrix from the residuals uses a weight factor.

    The DIIS extrapolation is followed by a Kerker mixing step.

    The code is m_diis.F90. The DIIS history is kept in a circular stack, implemented using the new framework for reference-counted types. This might be overkill for this particular use, and there are a few rough edges, but it works well.

The default convergence criteria remains based on the differences in the density matrix, but in this case the differences are from step to step, not the more fundamental `DM_out-DM_in`. Perhaps some other criterion should be made the default (max $|\Delta rho(G)|$, convergence of the free-energy...)

Note that with charge mixing the Harris energy as it is currently computed in Siesta loses its meaning, since there is no `DM_in`. The program prints zeroes in the Harris energy field.

Note that the KS energy is correctly computed throughout the scf cycle, as there is an extra step for the calculation of the charge stemming from `DM_out`, which also updates the energies. Forces and

final energies are correctly computed with the final `DM_out`, regardless of the setting of the option for mixing after scf convergence.

Initial tests suggest that charge mixing has some desirable properties and could be a drop-in replacement for density-matrix mixing, but many more tests are needed to calibrate its efficiency for different kinds of systems, and the heuristics for the (perhaps too many) parameters:

SCF.Kerker.q0sq 0 Ry *(energy)*

Determines the parameter q_0^2 featuring in the Kerker preconditioning, which is always performed on all components of $\rho(\mathbf{G})$, even those treated with the DIIS scheme.

SCF.RhoGMixingCutoff 9 Ry *(energy)*

Determines the sub-set of G vectors which will undergo the DIIS procedure. Only those with kinetic energies below this cutoff will be considered. The optimal extrapolation of the $\rho(\mathbf{G})$ elements will be replaced in the fourier series before performing the Kerker mixing.

SCF.RhoG.DIIS.Depth **0** *(integer)*

Determines the maximum number of previous steps considered in the DIIS procedure.

NOTE: The information from the first scf step is not included in the DIIS history. There is no provision yet for any other kind of "kick-starting" procedure. The logic is in m_rhog (rhog_mixing routine).

SCF.RhoG.Metric.Preconditioner.Cutoff ⟨**None**⟩ *(energy)*

Determines the value of q_1^2 in the weighing of the different $\mathbf{G}$ components in the scalar products among residuals in the DIIS procedure. Following the KF ansatz, this parameter is chosen so that the smallest (non-zero) $\mathbf{G}$ has a weight 20 times larger than that of the smallest G vector in the DIIS set.

The default is the result of the KF prescription.

SCF.DebugRhoGMixing **false** *(logical)*

Controls the level of debugging output in the mixing procedure (basically whether the first few stars worth of Fourier components are printed). Note that this feature will only display the components in the master node.

Debug.DIIS **false** *(logical)*

Controls the level of debugging output in the DIIS procedure. If set, the program prints the DIIS matrix and the extrapolation coefficients.

SCF.MixCharge.SCF1 **false** *(logical)*

Logical variable to indicate whether or not the charge is mixed in the first SCF cycle. Anecdotal evidence indicates that it might be advantageous, at least for calculations started from scratch, to avoid that first mixing, and retain the "out" charge density as "in" for the next step.

6.9.4 Initialization of the density-matrix

NOTE: The conditions and options for density-matrix re-use are quite varied and not completely orthogonal at this point. For further information, see routine `Src/m_new_dm.F`. What follows is a summary.

The Density matrix can be:

```
1. Synthesized directly from atomic occupations.
   (See the options below for spin considerations)
2. Read from a .DM file (if the appropriate options are set)
3. Extrapolated from previous geometry steps
   (this includes as a special case the re-use of the DM
    of the previous geometry iteration)

In cases 2 and 3, the structure of the read or extrapolated DM
is automatically adjusted to the current sparsity pattern.

In what follows, "Initialization" of the DM means that the DM is
either read from file (if available) or synthesized from atomic
data. This is confusing, and better terminology should be used.


Special cases:

       Harris functional: The matrix is always initialized

       Force calculation: The DM should be written to disk
                          at the time of the "no displacement"
                          calculation and read from file at
                          every subsequent step.

       Variable-cell calculation:

         If the auxiliary cell changes, the DM is forced to be
         synthesized (conceivably one could rescue some important
         information from an old DM, but it is too much trouble
         for now). NOTE that this is a change in policy with respect
         to previous versions of the program, in which a (blind?)
         re-use was allowed, except if 'ReInitialiseDM' was 'true'.
         Now 'ReInitialiseDM' is 'true' by default. Setting it to
         'false' is not recommended.

         In all other cases (including "server operation"), the
         default is to allow DM re-use (with possible extrapolation)
         from previous geometry steps.

         For "CG" calculations, the default is not to extrapolate the
         DM (unless requested by setting 'DM.AllowExtrapolation' to
         "true"). The previous step's DM is reused.

         The fdf variables 'DM.AllowReuse' and 'DM.AllowExtrapolation'
         can be used to turn off DM re-use and extrapolation.
```

DM.UseSaveDM **false** *(logical)*

Instructs to read the density matrix stored in file `SystemLabel..DM` by a previous run.

SIESTA will continue even if `.DM` is not found.

NOTE: that if the spin settings has changed SIESTA allows reading a `.DM` from a similar calculation with different **Spin** option. This may be advantageous when going from non-polarized calculations to polarized, and beyond, see **Spin** for details.

DM.FormattedFiles **false** *(logical)*

Setting this alters the default for **DM.FormattedInput** and **DM.FormattedOutput**. Instructs to use formatted files for reading and writing the density matrix. In this case, the files are labelled `SystemLabel.DMF`.

Only usable if one has problems transferring files from one computer to another.

DM.FormattedInput **false** *(logical)*

Instructs to use formatted files for reading the density matrix.

DM.FormattedOutput **false** *(logical)*

Instructs to use formatted files for writing the density matrix.

DM.InitSpin.AF **false** *(logical)*

It defines the initial spin density for a spin polarized calculation. The spin density is initially constructed with the maximum possible spin polarization for each atom in its atomic configuration. This variable defines the relative orientation of the atomic spins:

If **false** the initial spin-configuration is a ferromagnetic order (all spins up). If **true** all odd atoms are initialized to spin-up, all even atoms are initialized to spin-down.

%block DM.InitSpin ⟨**None**⟩ *(block)*

Define the initial spin density for a spin polarized calculation atom by atom. In the block there is one line per atom to be spin-polarized, containing the atom index (integer, ordinal in the block **AtomicCoordinatesAndAtomicSpecies**) and the desired initial spin-polarization (real, positive for spin up, negative for spin down). A value larger than possible will be reduced to the maximum possible polarization, keeping its sign. Maximum polarization can also be given by introducing the symbol `+` or `-` instead of the polarization value. There is no need to include a line for every atom, only for those to be polarized. The atoms not contemplated in the block will be given non-polarized initialization.

For non-collinear spin, the spin direction may be specified for each atom by the polar angle θ and the azimuthal angle ϕ (using the physics ISO convention), given as the last two arguments in degrees. If not specified, $\theta = 0$ is assumed (z-polarized). **Spin** must be set to use non-collinear or spin-orbit for the directions to have effect.

Example:

```
%block DM.InitSpin
   5  -1.   90.   0.   # Atom index, spin, theta, phi (deg)
   3   +    45. -90.
   7   -
%endblock DM.InitSpin
```

In the above example, atom 5 is polarized in the x-direction.

If this block is defined, but empty, all atoms are not polarized. This block has precedence over **DM.InitSpinAF**.

DM.AllowReuse **true** *(logical)*

Controls whether density matrix information from previous geometry iterations is re-used to start the new geometry's SCF cycle.

DM.AllowExtrapolation **true** *(logical)*

Controls whether the density matrix information from several previous geometry iterations is extrapolated to start the new geometry's SCF cycle. This feature is useful for molecular dynamics simulations and possibly also for geometry relaxations. The number of geometry steps saved is controlled by the variable **DM.History.Depth**.

This is default **true** for molecular-dynamics simulations, but **false**, for now, for geometry-relaxations (pending further tests which users are kindly requested to perform).

DM.History.Depth **1** *(integer)*

Sets the number of geometry steps for which density-matrix information is saved for extrapolation.

6.9.5 Initialization of the SCF cycle with charge densities

SCF.Read.Charge.NetCDF **false** *(logical)*

Instructs SIESTA to read the charge density stored in the netCDF file `Rho.IN.grid.nc`. This feature allows the easier re-use of electronic-structure information from a previous run. It is not necessary that the basis sets are "similar" (a requirement if density-matrices are to be read in).

NOTE: this is an experimental feature. Until robust checks are implemented, care must be taken to make sure that the FFT grids in the `.grid.nc` file and in SIESTA are the same.

SCF.Read.Deformation.Charge.NetCDF **false** *(logical)*

Instructs Siesta to read the deformation charge density stored in the netCDF file `DeltaRho.IN.grid.nc`. This feature allows the easier re-use of electronic-structure information from a previous run. It is not necessary that the basis sets are "similar" (a requirement if density-matrices are to be read in). The deformation charge is particularly useful to give a good starting point for slightly different geometries.

NOTE: this is an experimental feature. Until robust checks are implemented, care must be taken to make sure that the FFT grids in the `.grid.nc` file and in Siesta are the same.

6.9.6 Output of density matrix and Hamiltonian

Performance Note: For large-scale calculations, writing the DM at every scf step can have a severe impact on performance. The sparse-matrix I/O is undergoing a re-design, to facilitate the analysis of data and to increase the efficiency.

Use.Blocked.WriteMat **false** *(logical)*

By using blocks of orbitals (according to the underlying default block-cyclic distribution), the sparse-matrix I/O can be speeded-up significantly, both by saving MPI communication and by

reducing the number of file accesses. This is essential for large systems, for which the I/O could take a significant fraction of the total computation time.

To enable this "blocked format" (recommended for large-scale calculations) use the option **Use.Blocked.WriteMat true**. Note that it is off by default.

The new format is not backwards compatible. A converter program (`Util/DensityMatrix/dmUnblock.F90`) has been written to post-process those files intended for further analysis or re-use in Siesta. This is the best option for now, since it allows liberal checkpointing with a much smaller time consumption, and only incurs costs when re-using or analyzing files.

Note that TranSIESTA will continue to produce `SystemLabel.DM` files, in the old format (See save_density_matrix.F)

To test the new features, the option **S.Only true** can be used. It will produce three files: a standard one, another one with optimized MPI communications, and a third, blocked one.

Write.DM **true** *(logical)*

Control the creation of the current iterations density matrix to a file for restart purposes and post-processing. If **false** nothing will be written.

If **Use.Blocked.WriteMat** is **false** the `SystemLabel.DM` file will be written. Otherwise these density matrix files will be created; `DM_MIXED.blocked` and `DM_OUT.blocked` which are the mixed and the diagonalization output, respectively.

Write.DM.end.of.cycle ⟨**Write.DM**⟩ *(logical)*

Equivalent to **Write.DM**, but will only write at the end of each SCF loop.

NOTE: the file generated depends on **SCF.MixAfterConvergence**.

Write.H **false** *(logical)*

Whether restart Hamiltonians should be written (not intrinsically supported in 4.1).

If **true** these files will be created; `H_MIXED` or `H_DMGEN` which is the mixed or the generated Hamiltonian from the current density matrix, respectively. If **Use.Blocked.WriteMat** the just mentioned files will have the additional suffix **.blocked**.

Write.H.end.of.cycle ⟨**Write.H**⟩ *(logical)*

Equivalent to **Write.H**, but will only write at the end of each SCF loop.

NOTE: the file generated depends on **SCF.MixAfterConvergence**.

The following options control the creation of netCDF files. The relevant routines have not been optimized yet for large-scale calculations, so in this case the options should not be turned on (they are off by default).

Write.DM.NetCDF **true** *(logical)*

It determines whether the density matrix (after the mixing step) is output as a `DM.nc` netCDF file or not.

The file is overwritten at every SCF step. Use the **Write.DM.History.NetCDF** option if a complete history is desired.

The `DM.nc` and standard DM file formats can be converted at will with the programs in `Util/DensityMatrix` directory. Note that the DM values in the `DM.nc` file are in single precision.

Write.DMHS.NetCDF **true** *(logical)*

If true, the input density matrix, Hamiltonian, and output density matrix, are stored in a netCDF file named `DMHS.nc`. The file also contains the overlap matrix S.

The file is overwritten at every SCF step. Use the **Write.DMHS.History.NetCDF** option if a complete history is desired.

Write.DM.History.NetCDF **false** *(logical)*

If **true**, a series of netCDF files with names of the form `DM-NNNN.nc` is created to hold the complete history of the density matrix (after mixing). (See also **Write.DM.NetCDF**). Each file corresponds to a geometry step.

Write.DMHS.History.NetCDF **false** *(logical)*

If **true**, a series of netCDF files with names of the form `DMHS-NNNN.nc` is created to hold the complete history of the input and output density matrix, and the Hamiltonian. (See also **Write.DMHS.NetCDF**). Each file corresponds to a geometry step. The overlap matrix is stored only once per SCF cycle.

Write.TSHS.History **false** *(logical)*

If true, a series of TSHS files with names of the form `SystemLabel.N.TSHS` is created to hold the complete history of the Hamiltonian and overlap matrix. Each file corresponds to a geometry step. The overlap matrix is stored only once per SCF cycle. This option only works with TranSIESTA.

6.9.7 Convergence criteria

NOTE: The older options with a **DM** prefix is still working for backwards compatibility. However, the following flags has precedence.

Note that all convergence criteria are additive and may thus be used simultaneously for complete control.

SCF.DM.Converge **true** *(logical)*

Logical variable to use the density matrix elements as monitor of self-consistency.

SCF.DM.Tolerance 10^{-4} *(real)*

depends on: **SCF.DM.Converge**

Tolerance of Density Matrix. When the maximum difference between the output and the input on each element of the DM in a SCF cycle is smaller than **SCF.DM.Tolerance**, the self-consistency has been achieved.

NOTE: DM.Tolerance is the actual default for this flag.

DM.Normalization.Tolerance 10^{-5} *(real)*

Tolerance for unnormalized density matrices (typically the product of solvers such as PEXSI which have a built-in electron-count tolerance). If this tolerance is exceeded, the program stops. It is understood as a fractional tolerance. For example, the default will allow an excess or shorfall of 0.01 electrons in a 1000-electron system.

SCF.H.Converge **true** *(logical)*

Logical variable to use the Hamiltonian matrix elements as monitor of self-consistency: this is considered achieved when the maximum absolute change (dHmax) in the H matrix elements is below **SCF.H.Tolerance**. The actual meaning of dHmax depends on whether DM or H mixing is in effect: if mixing the DM, dHmax refers to the change in H(in) with respect to the previous step; if mixing H, dHmax refers to H(out)-H(in) in the previous(?) step.

SCF.H.Tolerance 10^{-3} eV *(energy)*

depends on: **SCF.H.Converge**

If **SCF.H.Converge** is **true**, then self-consistency is achieved when the maximum absolute change in the Hamiltonian matrix elements is below this value.

SCF.EDM.Converge **true** *(logical)*

Logical variable to use the energy density matrix elements as monitor of self-consistency: this is considered achieved when the maximum absolute change (dEmax) in the energy density matrix elements is below **SCF.EDM.Tolerance**. The meaning of dEmax is equivalent to that of **SCF.DM.Tolerance**.

SCF.EDM.Tolerance 10^{-3} eV *(energy)*

depends on: **SCF.EDM.Converge**

If **SCF.EDM.Converge** is **true**, then self-consistency is achieved when the maximum absolute change in the energy density matrix elements is below this value.

SCF.FreeE.Converge **false** *(logical)*

Logical variable to request an additional requirement for self-consistency: it is considered achieved when the change in the total (free) energy between cycles of the SCF procedure is below **SCF.FreeE.Tolerance** and the density matrix change criterion is also satisfied.

SCF.FreeE.Tolerance 10^{-4} eV *(energy)*

depends on: **SCF.FreeE.Converge**

If **SCF.FreeE.Converge** is **true**, then self-consistency is achieved when the change in the total (free) energy between cycles of the SCF procedure is below this value and the density matrix change criterion is also satisfied.

SCF.Harris.Converge **false** *(logical)*

Logical variable to use the Harris energy as monitor of self-consistency: this is considered achieved when the change in the Harris energy between cycles of the SCF procedure is below **SCF.Harris.Tolerance**. This is useful if only energies are needed, as the Harris energy tends to converge faster than the Kohn-Sham energy. The user is responsible for using the correct energies in further processing, e.g., the Harris energy if the Harris criterion is used.

To help in basis-optimization tasks, a new file `BASIS_HARRIS_ENTHALPY` is provided, holding the same information as `BASIS_ENTHALPY` but using the Harris energy instead of the Kohn-Sham energy.

NOTE: setting this to **true** makes **SCF.DM.Converge SCF.H.Converge** default to **false**.

SCF.Harris.Tolerance 10^{-4} eV *(energy)*

depends on: **SCF.Harris.Converge**

If **SCF.Harris.Converge** is **true**, then self-consistency is achieved when the change in the Harris energy between cycles of the SCF procedure is below this value. This is useful if only

energies are needed, as the Harris energy tends to converge faster than the Kohn-Sham energy.

6.10 The real-space grid and the eggbox-effect

SIESTA uses a finite 3D grid for the calculation of some integrals and the representation of charge densities and potentials. Its fineness is determined by its plane-wave cutoff, as given by the **MeshCutoff** option. It means that all periodic plane waves with kinetic energy lower than this cutoff can be represented in the grid without aliasing. In turn, this implies that if a function (e.g. the density or the effective potential) is an expansion of only these plane waves, it can be Fourier transformed back and forth without any approximation.

The existence of the grid causes the breaking of translational symmetry (the egg-box effect, due to the fact that the density and potential *do have* plane wave components above the mesh cutoff). This symmetry breaking is clear when moving one single atom in an otherwise empty simulation cell. The total energy and the forces oscillate with the grid periodicity when the atom is moved, as if the atom were moving on an eggbox. In the limit of infinitely fine grid (infinite mesh cutoff) this effect disappears.

For reasonable values of the mesh cutoff, the effect of the eggbox on the total energy or on the relaxed structure is normally unimportant. However, it can affect substantially the process of relaxation, by increasing the number of steps considerably, and can also spoil the calculation of vibrations, usually much more demanding than relaxations.

The `Util/Scripting/eggbox_checker.py` script can be used to diagnose the eggbox effect to be expected for a particular pseudopotential/basis-set combination.

Apart from increasing the mesh cutoff (see the **MeshCutoff** option), the following options might help in lessening a given eggbox problem. But note also that a filtering of the orbitals and the relevant parts of the pseudopotential and the pseudocore charge might be enough to solve the issue (see Sec. 6.3.9).

MeshCutoff 100 Ry *(energy)*

Defines the plane wave cutoff for the grid.

MeshSubDivisions 2 *(integer)*

Defines the number of sub-mesh points in each direction used to save index storage on the mesh. It affects the memory requirements and the CPU time, but not the results.

NOTE: the default value might be a bit conservative. Users might experiment with higher values, 4 or 6, to lower the memory and cputime usage.

%block Grid.CellSampling ⟨**None**⟩ *(block)*

It specifies points within the grid cell for a symmetrization sampling.

For a given grid the grid-cutoff convergence can be improved (and the eggbox lessened) by recovering the lost symmetry: by symmetrizing the sensitive quantities. The full symmetrization implies an integration (averaging) over the grid cell. Instead, a finite sampling can be performed.

It is a sampling of rigid displacements of the system with respect to the grid. The original grid-system setup (one point of the grid at the origin) is always calculated. It is the (0,0,0) displacement. The block **Grid.CellSampling** gives the additional displacements wanted for

the sampling. They are given relative to the grid-cell vectors, i.e., (1,1,1) would displace to the next grid point across the body diagonal, giving an equivalent grid-system situation (a useless displacement for a sampling).

Examples: Assume a cubic cell, and therefore a (smaller) cubic grid cell. If there is no block or the block is empty, then the original (0,0,0) will be used only. The block:

```
%block Grid.CellSampling
   0.5    0.5    0.5
%endblock Grid.CellSampling
```

would use the body center as a second point in the sampling. Or:

```
%block Grid.CellSampling
   0.5    0.5    0.0
   0.5    0.0    0.5
   0.0    0.5    0.5
%endblock Grid.CellSampling
```

gives an fcc kind of sampling, and

```
%block Grid.CellSampling
   0.5    0.0    0.0
   0.0    0.5    0.0
   0.0    0.0    0.5
   0.0    0.5    0.5
   0.5    0.0    0.5
   0.5    0.5    0.0
   0.5    0.5    0.5
%endblock Grid.CellSampling
```

gives again a cubic sampling with half the original side length. It is not trivial to choose a right set of displacements so as to maximize the new 'effective' cutoff. It depends on the kind of cell. It may be automatized in the future, but it is now left to the user, who introduces the displacements manually through this block.

The quantities which are symmetrized are: (*i*) energy terms that depend on the grid, (*ii*) forces, (*iii*) stress tensor, and (*iv*) electric dipole.

The symmetrization is performed at the end of every SCF cycle. The whole cycle is done for the (0,0,0) displacement, and, when the density matrix is converged, the same (now fixed) density matrix is used to obtain the desired quantities at the other displacements (the density matrix itself is *not* symmetrized as it gives a much smaller egg-box effect). The CPU time needed for each displacement in the **Grid.CellSampling** block is of the order of one extra SCF iteration.

This may be required in systems where very precise forces are needed, and/or if partial cores are used. It is advantageous to test whether the forces are sampled sufficiently by sampling one point.

Additionally this may be given as a list of 3 integers which corresponds to a "Monkhorst-Pack" like grid sampling. I.e.

```
Grid.CellSampling [2 2 2]
```

is equivalent to

```
%block Grid.CellSampling
   0.5    0.0    0.0
```

```
  0.0    0.5    0.0
  0.5    0.5    0.0
  0.0    0.0    0.5
  0.5    0.0    0.5
  0.0    0.5    0.5
  0.5    0.5    0.5
%endblock Grid.CellSampling
```

This is an easy method to see if the flag is important for your system or not.

%block EggboxRemove ⟨**None**⟩ *(block)*

For recovering translational invariance in an approximate way.

It works by substracting from Kohn-Sham's total energy (and forces) an approximation to the eggbox energy, sum of atomic contributions. Each atom has a predefined eggbox energy depending on where it sits on the cell. This atomic contribution is species dependent and is obviously invariant under grid-cell translations. Each species contribution is thus expanded in the appropriate Fourier series. It is important to have a smooth eggbox, for it to be represented by a few Fourier components. A jagged egg-box (unless very small, which is then unimportant) is often an indication of a problem with the pseudo.

In the block there is one line per Fourier component. The first integer is for the atomic species it is associated with. The other three represent the reciprocal lattice vector of the grid cell (in units of the basis vectors of the reciprocal cell). The real number is the Fourier coefficient in units of the energy scale given in **EggboxScale** (see below), normally 1 eV.

The number and choice of Fourier components is free, as well as their order in the block. One can choose to correct only some species and not others if, for instance, there is a substantial difference in hardness of the cores. The 0 0 0 components will add a species-dependent constant energy per atom. It is thus irrelevant except if comparing total energies of different calculations, in which case they have to be considered with care (for instance by putting them all to zero, i.e. by not introducing them in the list). The other components average to zero representing no bias in the total energy comparisons.

If the total energies of the free atoms are put as 0 0 0 coefficients (with spin polarisation if adequate etc.) the corrected total energy will be the cohesive energy of the system (per unit cell).

Example: For a two species system, this example would give a quite sufficent set in many instances (the actual values of the Fourier coefficients are not realistic).

```
%block EggBoxRemove
  1   0   0   0 -143.86904
  1   0   0   1    0.00031
  1   0   1   0    0.00016
  1   0   1   1   -0.00015
  1   1   0   0    0.00035
  1   1   0   1   -0.00017
  2   0   0   0 -270.81903
  2   0   0   1    0.00015
  2   0   1   0    0.00024
  2   1   0   0    0.00035
  2   1   0   1   -0.00077
  2   1   1   0   -0.00075
  2   1   1   1   -0.00002
```

```
%endblock EggBoxRemove
```

It represents an alternative to grid-cell sampling (above). It is only approximate, but once the Fourier components for each species are given, it does not represent any computational effort (neither memory nor time), while the grid-cell sampling requires CPU time (roughly one extra SCF step per point every MD step).

It will be particularly helpful in atoms with substantial partial core or semicore electrons.

NOTE: this should only be used for fixed cell calculations, i.e. not with **MD.VariableCell**.

For the time being, it is up to the user to obtain the Fourier components to be introduced. They can be obtained by moving one isolated atom through the cell to be used in the calculation (for a give cell size, shape and mesh), once for each species. The Util/Scripting/eggbox_checker.py script can be used as a starting point for this.

EggboxScale 1 eV *(energy)*

Defines the scale in which the Fourier components of the egg-box energy are given in the **EggboxRemove** block.

6.11 Matrix elements of the Hamiltonian and overlap

NeglNonOverlapInt **false** *(logical)*

Logical variable to neglect or compute interactions between orbitals which do not overlap. These come from the KB projectors. Neglecting them makes the Hamiltonian more sparse, and the calculation faster.

NOTE: use with care!

SCF.Write.Extra **false** *(logical)*

Instructs SIESTA to write out a variety of files with the Hamiltonian and density matrix.

The output depends on whether a Hamiltonian mixing or density matrix mixing is performed (see **SCF.Mixing**).

These files are created

- `H_MIXED`; the Hamiltonian after mixing
- `DM_OUT`; the density matrix as calculated by the current iteration
- `H_DMGEN`; the Hamiltonian used to calculate the density matrix
- `DM_MIXED`; the density matrix after mixing

SaveHS **false** *(logical)*

Instructs to write the Hamiltonian and overlap matrices, as well as other data required to generate bands and density of states, in file `SystemLabel.HSX`. The `.HSX` format is more compact than the traditional `.HS`, and the Hamiltonian, overlap matrix, and relative-positions array (which is always output, even for gamma-point only calculations) are in single precision.

The program `hsx2hs` in `Util/HSX` can be used to generate an old-style `.HS` file if needed.

SIESTA produces also an `.HSX` file if the **COOP.Write** option is active.

See also the **Write.DMHS.NetCDF** and **Write.DMHS.History.NetCDF** options.

6.11.1 The auxiliary supercell

When using k-points, this auxiliary supercell is needed to compute properly the matrix elements involving orbitals in different unit cells. It is computed automatically by the program at every geometry step.

FixAuxiliaryCell **false** *(logical)*

Logical variable to control whether the auxiliary cell is changed during a variable cell optimization.

NaiveAuxiliaryCell **false** *(logical)*

If **true**, the program does not check whether the auxiliary cell constructed with a naive algorithm is appropriate. This variable is only useful if one wishes to reproduce calculations done with previous versions of the program in which the auxiliary cell was not large enough, as indicated by warnings such as:

```
WARNING: orbital pair 1 341 is multiply connected
```

Only small numerical differences in the results are to be expected.

Note that for gamma-point-only calculations there is an implicit "folding" of matrix elements corresponding to the images of orbitals outside the unit cell. If information about the specific values of these matrix elements is needed (as for COOP/COHP analysis), one has to make sure that the unit cell is large enough.

6.12 Calculation of the electronic structure

SIESTA can use three qualitatively different methods to determine the electronic structure of the system. The first is standard diagonalization, which works for all systems and has a cubic scaling with the size. The second is based on the direct minimization of a special functional over a set of trial orbitals. These orbitals can either extend over the entire system, resulting in a cubic scaling algorithm, or be constrained within a localization radius, resulting in a linear scaling algorithm. The former is a recent implementation (described in 6.12.4), that can be viewed as an equivalent approach to diagonalization in terms of the accuracy of the solution; the latter is the historical O(N) method used by SIESTA (described in 6.12.5); it scales in principle linearly with the size of the system (only if the size is larger than the radial cutoff for the local solution wave-functions), but is quite fragile and substantially more difficult to use, and only works for systems with clearly separated occupied and empty states. The default is to use diagonalization. The third method (PEXSI) is based on the pole expansion of the Fermi-Dirac function and the direct computation of the density matrix via an efficient scheme of selected inversion (see Sec 6.13).

The calculation of the H and S matrix elements is always done with an O(N) method. The actual scaling is not linear for small systems, but it becomes O(N) when the system dimensions are larger than the scale of orbital r_c's.

The relative importance of both parts of the computation (matrix elements and solution) depends on the size and quality of the calculation. The mesh cutoff affects only the matrix-element calculation; orbital cutoff radii affect the matrix elements and all solvers except diagonalization; the need for **k**-point sampling affects the solvers only, and the number of basis orbitals affects them all.

In practice, the vast majority of users employ diagonalization (or the OMM method) for the cal-

culation of the electronic structure. This is so because the vast majority of calculations (done for intermediate system sizes) would not benefit from the O(N) or PEXSI solvers.

SolutionMethod **diagon** *(string)*

Character string to choose among diagonalization (**diagon**), cubic-scaling minimization (**OMM**), Order-N (**OrderN**) solution of the Kohn-Sham Hamiltonian, **transiesta**, or the PEXSI method (**PEXSI**).

6.12.1 Diagonalization options

NumberOfEigenStates ⟨**all orbitals**⟩ *(integer)*

This parameter allows the user to reduce the number of eigenstates that are calculated from the maximum possible. The benefit is that, for a gamma point calculation, the cost of the diagonalisation is reduced by finding fewer eigenvectors. For example, during a geometry optimisation, only the occupied states are required rather than the full set of virtual orbitals. Note, that if the electronic temperature is greater than zero then the number of partially occupied states increases, depending on the band gap. The value specified must greater than the number of occupied states and less than the number of basis functions.

Diag.ELPA **false** *(logical)*

(For parallel gamma-point calculations without spin orbit only) Use the ELPA routines for diagonalization. Specifying a number of eigenvectors to store is possible through the symbol **NumberOfEigenStates** (see above).

A description of some algorithms present in ELPA can be found in:

T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations", Parallel Computing 37, 783-794 (2011). doi:10.1016/j.parco.2011.05.002.

Marek, A.; Blum, V.; Johanni, R.; Havu, V.; Lang, B.; Auckenthaler, T.; Heinecke, A.; Bungartz, H.-J.; Lederer, H. "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science", Journal of Physics Condensed Matter, 26 (2014) doi:10.1088/0953-8984/26/21/213201

NOTE: It is not compatible with the **Diag.ParallelOverK** option.

Diag.MRRR **false** *(logical)*

For parallel Γ-point calculations without spin orbit only.

Use the MRRR method in ScaLAPACK for diagonalization. Specifying a number of eigenvectors to store is possible through the symbol **NumberOfEigenStates** (see above).

NOTE: it is not compatible with the **Diag.ParallelOverK** option.

NOTE: The MRRR method is defaulted not to be compiled in, however, if your ScaLAPACK library does contain the relevant sources one may add this pre-processor flag `-DSIESTA__MRRR`.

Diag.UseNewDiagk **false** *(logical)*

Selects whether a more efficient diagonalization routine (with intermediate storage of eigenvectors in NetCDF format) is used for the case of k-point sampling.

In order to use the new routine, netCDF support should be compiled in. Specifying a number

of eigenvectors to store is possible through the symbol **NumberOfEigenStates** (see above). Note that for now, for safety, all eigenvectors for a given k-point and spin are computed by the diagonalization routine, but only that number specified by the user are stored. If they are insufficient, the program stops. A rule of thumb to select the number of eigenvectors to store is to count the number of electrons and divide by two, and then apply a "safety factor" of around 1.1-1.2 to take into account fractional occupations and band overlaps.

A new file `OCCS` is produced with information about the number of states occupied.

This is an experimental feature.

NOTE: It is not compatible with the **Diag.ParallelOverK** option.

Diag.DivideAndConquer **true** *(logical)*

Logical to select whether the normal or Divide and Conquer algorithms are used within the Lapack diagonalisation routines.

Diag.AllInOne **false** *(logical)*

Logical to select whether a single call to lapack/scalapack is made to perform the diagonalisation or whether the individual steps are controlled by SIESTA. Normally this option should not need to be used.

Diag.NoExpert **false** *(logical)*

Logical to select whether the simple or expert versions of the lapack/ scalapack routines are used. Usually the expert routines are faster, but may require slightly more memory.

Diag.PreRotate **false** *(logical)*

Logical to select whether the eigensystem is transformed according to previously saved eigenvectors to create a near diagonal matrix and then back transformed afterwards. This is included for future options, but currently should not make any difference except to increase the computational work!

Diag.Use2D **true** *(logical)*

Logical to select whether a 1-D or 2-D data decomposition should be used when calling scalapack. The use of 2-D leads to superior scaling to large numbers of processors and is therefore the default. This option only influences the parallel performance.

6.12.2 Output of eigenvalues and wavefunctions

This section focuses on the output of eigenvalues and wavefunctions produced during the (last) iteration of the self-consistent cycle, and associated to the appropriate k-point sampling.

For band-structure calculations (which typically use a different set of k-points) and specific requests for wavefunctions, see Secs. 6.14 and 6.15, respectively.

The complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to a NetCDF file `WFS.nc` if the **Diag.UseNewDiagk** option is in effect.

The complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to `SystemLabel.fullBZ.WFSX` if the **COOP.Write** option is in effect.

WriteEigenvalues **false** *(logical)*

If **true** it writes the Hamiltonian eigenvalues for the sampling $\vec{k}$ points, in the main output file. If **false**, it writes them in the file `SystemLabel.EIG`, which can be used by the `Eig2DOS` postprocessing utility (in the Util/Eig2DOS directory) for obtaining the density of states.

NOTE: this option only works for **SolutionMethod** which calculates the eigenvalues.

6.12.3 Occupation of electronic states and Fermi level

OccupationFunction FD *(string)*

String variable to select the function that determines the occupation of the electronic states. Two options are available:

FD The usual Fermi-Dirac occupation function is used.

MP The occupation function proposed by Methfessel and Paxton (Phys. Rev. B, **40**, 3616 (1989)), is used.

The smearing of the electronic occupations is done, in both cases, using an energy width defined by the **ElectronicTemperature** variable. Note that, while in the case of Fermi-Dirac, the occupations correspond to the physical ones if the electronic temperature is set to the physical temperature of the system, this is not the case in the Methfessel-Paxton function. In this case, the tempeature is just a mathematical artifact to obtain a more accurate integration of the physical quantities at a lower cost. In particular, the Methfessel-Paxton scheme has the advantage that, even for quite large smearing temperatures, the obtained energy is very close to the physical energy at $T = 0$. Also, it allows a much faster convergence with respect to k-points, specially for metals. Finally, the convergence to selfconsistency is very much improved (allowing the use of larger mixing coefficients).

For the Methfessel-Paxton case, one can use relatively large values for the **ElectronicTemperature** parameter. How large depends on the specific system. A guide can be found in the article by J. Kresse and J. Furthmüller, Comp. Mat. Sci. **6**, 15 (1996).

If Methfessel-Paxton smearing is used, the order of the corresponding Hermite polynomial expansion must also be chosen (see description of variable **OccupationMPOrder**).

We finally note that, in both cases (FD and MP), once a finite temperature has been chosen, the relevant energy is not the Kohn-Sham energy, but the Free energy. In particular, the atomic forces are derivatives of the Free energy, not the KS energy. See R. Wentzcovitch *et al.*, Phys. Rev. B **45**, 11372 (1992); S. de Gironcoli, Phys. Rev. B **51**, 6773 (1995); J. Kresse and J. Furthmüller, Comp. Mat. Sci. **6**, 15 (1996), for details.

OccupationMPOrder 1 *(integer)*

Order of the Hermite-Gauss polynomial expansion for the electronic occupation functions in the Methfessel-Paxton scheme (see Phys. Rev. B **40**, 3616 (1989)). Specially for metals, higher order expansions provide better convergence to the ground state result, even with larger smearing temperatures, and provide also better convergence with k-points.

NOTE: only used if **OccupationFunction** is **MP**.

ElectronicTemperature 300 K *(temperature/energy)*

Temperature for Fermi-Dirac or Methfessel-Paxton distribution. Useful specially for metals, and to accelerate selfconsistency in some cases.

6.12.4 Orbital minimization method (OMM)

The OMM is an alternative cubic-scaling solver that uses a minimization algorithm instead of direct diagonalization to find the occupied subspace. The main advantage over diagonalization is the possibility of iteratively reusing the solution from each SCF/MD step as the starting guess of the following one, thus greatly reducing the time to solution. Typically, therefore, the first few SCF cycles of the first MD step of a simulation will be slower than diagonalization, but the rest will be faster. The main disadvantages are that individual Kohn-Sham eigenvalues are not computed, and that only a fixed, integer number of electrons at each k point/spin is allowed. Therefore, only spin-polarized calculations with **Spin.Fix** are allowed, and **Spin.Total** must be chosen appropriately. For non-Γ point calculations, the number of electrons is set to be equal at all k points. Non-collinear calculations (see **Spin**) are not supported at present.

It is important to note that the OMM requires all occupied Kohn-Sham eigenvalues to be negative; this can be achieved by applying a shift to the eigenspectrum, controlled by **ON.eta** (in this case, **ON.eta** simply needs to be higher than the HOMO level). If the OMM exhibits a pathologically slow or unstable convergence, this is almost certainly due to the fact that the default value of **ON.eta** (**0.0 eV**) is too low, and should be raised by a few eV.

OMM.UseCholesky **true** *(logical)*

Select whether to perform a Cholesky factorization of the generalized eigenvalue problem; this removes the overlap matrix from the problem but also destroys the sparsity of the Hamiltonian matrix.

OMM.Use2D **true** *(logical)*

Select whether to use a 2D data decomposition of the matrices for parallel calculations. This generally leads to superior scaling for large numbers of MPI processes.

OMM.UseSparse **false** *(logical)*

Select whether to make use of the sparsity of the Hamiltonian and overlap matrices where possible when performing matrix-matrix multiplications (these operations are thus reduced from $O(N^3)$ to $O(N^2)$ without loss of accuracy).

NOTE: not compatible with **OMM.UseCholesky**, **OMM.Use2D**, or non-Γ point calculations

OMM.Precon **-1** *(integer)*

Number of SCF steps for *all* MD steps for which to apply a preconditioning scheme based on the overlap and kinetic energy matrices; for negative values the preconditioning is always applied. Preconditioning is usually essential for fast and accurate convergence (note, however, that it is not needed if a Cholesky factorization is performed; in such cases this variable will have no effect on the calculation).

NOTE: cannot be used with **OMM.UseCholesky**.

OMM.PreconFirstStep ⟨**OMM.Precon**⟩ *(integer)*

Number of SCF steps in the *first* MD step for which to apply the preconditioning scheme; if present, this will overwrite the value given in **OMM.Precon** for the first MD step only.

OMM.Diagon **0** *(integer)*

Number of SCF steps for *all* MD steps for which to use a standard diagonalization before switching to the OMM; for negative values diagonalization is always used, and so the calculation is effectively equivalent to **SolutionMethod diagon**. In general, selecting the first few SCF steps can speed up the calculation by removing the costly initial minimization (at present this works best for Γ point calculations).

OMM.DiagonFirstStep ⟨**OMM.Diagon**⟩ *(integer)*

Number of SCF steps in the *first* MD step for which to use a standard diagonalization before switching to the OMM; if present, this will overwrite the value given in **OMM.Diagon** for the first MD step only.

OMM.BlockSize ⟨**BlockSize**⟩ *(integer)*

Blocksize used for distributing the elements of the matrix over MPI processes. Specifically, this variable controls the dimension relating to the trial orbitals used in the minimization (equal to the number of occupied states at each k point/spin); the equivalent variable for the dimension relating to the underlying basis orbitals is controlled by **BlockSize**.

OMM.TPreconScale 10 Ry *(energy)*

Scale of the kinetic energy preconditioning (see C. K. Gan *et al.*, Comput. Phys. Commun. **134**, 33 (2001)). A smaller value indicates more aggressive kinetic energy preconditioning, while an infinite value indicates no kinetic energy preconditioning. In general, the kinetic energy preconditioning is much less important than the tensorial correction brought about by the overlap matrix, and so this value will have fairly little impact on the overall performace of the preconditioner; however, too aggressive kinetic energy preconditioning can have a detrimental effect on performance and accuracy.

OMM.RelTol 10^{-9} *(real)*

Relative tolerance in the conjugate gradients minimization of the Kohn-Sham band energy (see **ON.Etol**).

OMM.Eigenvalues **false** *(logical)*

Select whether to perform a diagonalization at the end of each MD step to obtain the Kohn-Sham eigenvalues.

OMM.WriteCoeffs **false** *(logical)*

Select whether to write the coefficients of the solution orbitals to file at the end of each MD step.

OMM.ReadCoeffs **false** *(logical)*

Select whether to read the coefficients of the solution orbitals from file at the beginning of a new calculation. Useful for restarting an interrupted calculation, especially when used in conjuction with **DM.UseSaveDM**. Note that the same number of MPI processes and values of **OMM.Use2D**, **OMM.BlockSize**, and **BlockSize** must be used when restarting.

OMM.LongOutput **false** *(logical)*

Select whether to output detailed information of the conjugate gradients minimization for each SCF step.

6.12.5 Order(N) calculations

The Ordern(N) subsystem is quite fragile and only works for systems with clearly separated occupied and empty states. Note also that the option to compute the chemical potential automatically does not yet work in parallel.

NOTE: Since it is used less often, bugs creeping into the O(N) solver have been more resilient than in more popular bits of the code. Work is ongoing to clean and automate the O(N) process, to make the solver more user-friendly and robust.

ON.functional **Kim** *(string)*

Choice of order-N minimization functionals:

Kim Functional of Kim, Mauri and Galli, PRB 52, 1640 (1995).

Ordejon-Mauri Functional of Ordejón et al, or Mauri et al, see PRB 51, 1456 (1995). The number of localized wave functions (LWFs) used must coincide with $N_{el}/2$ (unless spin polarized). For the initial assignment of LWF centers to atoms, atoms with even number of electrons, n, get $n/2$ LWFs. Odd atoms get $(n+1)/2$ and $(n-1)/2$ in an alternating sequence, ir order of appearance (controlled by the input in the atomic coordinates block).

files Reads localized-function information from a file and chooses automatically the functional to be used.

ON.MaxNumIter **1000** *(integer)*

Maximum number of iterations in the conjugate minimization of the electronic energy, in each SCF cycle.

ON.Etol 10^{-8} *(real)*

Relative-energy tolerance in the conjugate minimization of the electronic energy. The minimization finishes if $2(E_n - E_{n-1})/(E_n + E_{n-1}) \leq$ **ON.Etol**.

ON.eta 0 eV *(energy)*

Fermi level parameter of Kim *et al.*. This should be in the energy gap, and tuned to obtain the correct number of electrons. If the calculation is spin polarised, then separate Fermi levels for each spin can be specified.

ON.eta.alpha 0 eV *(energy)*

Fermi level parameter of Kim *et al.* for alpha spin electrons. This should be in the energy gap, and tuned to obtain the correct number of electrons. Note that if the Fermi level is not specified individually for each spin then the same global eta will be used.

ON.eta.beta 0 eV *(energy)*

Fermi level parameter of Kim *et al.* for beta spin electrons. This should be in the energy gap, and tuned to obtain the correct number of electrons. Note that if the Fermi level is not specified individually for each spin then the same global eta will be used.

ON.RcLWF 9.5 Bohr *(length)*

Localization redius for the Localized Wave Functions (LWF's).

ON.ChemicalPotential **false** *(logical)*

Specifies whether to calculate an order-N estimate of the Chemical Potential, by the projection method (Goedecker and Teter, PRB **51**, 9455 (1995); Stephan, Drabold and Martin, PRB **58**, 13472 (1998)). This is done by expanding the Fermi function (or density matrix) at a given temperature, by means of Chebyshev polynomials, and imposing a real space truncation on the density matrix. To obtain a realistic estimate, the temperature should be small enough (typically, smaller than the energy gap), the localization range large enough (of the order of the one you would use for the Localized Wannier Functions), and the order of the polynomial expansion sufficiently large (how large depends on the temperature; typically, 50-100).

NOTE: this option does not work in parallel. An alternative is to obtain the approximate value of the chemical potential using an initial diagonalization.

ON.ChemicalPotential.Use **false** *(logical)*

Specifies whether to use the calculated estimate of the Chemical Potential, instead of the parameter **ON.eta** for the order-N energy functional minimization. This is useful if you do not know the position of the Fermi level, typically in the beginning of an order-N run.

NOTE: this overrides the value of **ON.eta** and **ON.ChemicalPotential**. Also, this option does not work in parallel. An alternative is to obtain the approximate value of the chemical potential using an initial diagonalization.

ON.ChemicalPotential.Rc 9.5 Bohr *(length)*

Defines the cutoff radius for the density matrix or Fermi operator in the calculation of the estimate of the Chemical Potential.

ON.ChemicalPotential.Temperature 0.05 Ry *(temperature/energy)*

Defines the temperature to be used in the Fermi function expansion in the calculation of the estimate of the Chemical Potential. To have an accurate results, this temperature should be smaller than the gap of the system.

ON.ChemicalPotential.Order 100 *(integer)*

Order of the Chebishev expansion to calculate the estimate of the Chemical Potential.

ON.LowerMemory **false** *(logical)*

If **true**, then a slightly reduced memory algorithm is used in the 3-point line search during the order N minimisation. Only affects parallel runs.

Output of localized wavefunctions At the end of each conjugate gradient minimization of the energy functional, the LWF's are stored on disk. These can be used as an input for the same system in a restart, or in case something goes wrong. The LWF's are stored in sparse form in file SystemLabel.LWF

It is important to keep very good care of this file, since the first minimizations can take MANY steps. Loosing them will mean performing the whole minimization again. It is also a good practice to save it periodically during the simulation, in case a mid-run restart is necessary.

ON.UseSaveLWF **false** *(logical)*

Instructs to read the localized wave functions stored in file `SystemLabel.LWF` by a previous run.

6.13 The PEXSI solver

The PEXSI solver is based on the combination of the pole expansion of the Fermi-Dirac function and the computation of only a selected (sparse) subset of the elements of the matrices $(H - z_l S)^{-1}$ at each pole z_l.

This solver can efficiently use the sparsity pattern of the Hamiltonian and overlap matrices generated in SIESTA, and for large systems has a much lower computational complexity than that associated with the matrix diagonalization procedure. It is also highly scalable.

The PEXSI technique can be used to evaluate the electron density, free energy, atomic forces, density of states and local density of states without computing any eigenvalue or eigenvector of the Kohn-Sham Hamiltonian. It can achieve accuracy fully comparable to that obtained from a matrix diagonalization procedure for general systems, including metallic systems at low temperature.

The current implementation of the PEXSI solver in SIESTA makes use of the full fine-grained-level interface in the PEXSI library (`http://pexsi.org`), and can deal with spin-polarization, but it is still restricted to Γ-point calculations.

The following is a brief description of the input-file parameters relevant to the workings of the PEXSI solver. For more background, including a discussion of the conditions under which this solver is competitive, the user is referred to the paper [**?**], and references therein.

The technology involved in the PEXSI solver can also be used to compute densities of states and "local densities of states". These features are documented in this section and also linked to in the relevant general sections.

6.13.1 Pole handling

Note that the temperature for the Fermi-Dirac distribution which is pole-expanded is taken directly from the **ElectronicTemperature** parameter (see Sec. 6.12.3).

PEXSI.NumPoles **40** *(integer)*

Effective number of poles used to expand the Fermi-Dirac function.

PEXSI.deltaE 3 Ry *(energy)*

In principle **PEXSI.deltaE** should be $E_{\max} - \mu$, where $E_{\max}$ is the largest eigenvalue for (H,S), and μ is the chemical potential. However, due to the fast decay of the Fermi-Dirac function, **PEXSI.deltaE** can often be chosen to be much lower. In practice we set the default to be 3 Ryd. This number should be set to be larger if the difference between Tr[H·DM] and Tr[S∗EDM] (displayed in the output if **PEXSI.Verbosity** is at least 2) does not decrease with the increase of the number of poles.

PEXSI.Gap 0 Ry *(energy)*

Spectral gap. This can be set to be 0 in most cases.

6.13.2 Parallel environment and control options

MPI.Nprocs.SIESTA ⟨**total processors**⟩ *(integer)*

Specifies the number of MPI processes to be used in those parts of the program (such as

Hamiltonian setup and computation of forces) which are outside of the PEXSI solver itself. This is needed in large-scale calculations, for which the number of processors that can be used by the PEXSI solver is much higher than those needed by other parts of the code.

Note that when the PEXSI solver is not used, this parameter will simply reduce the number of processors actually used by all parts of the program, leaving the rest idle for the whole calculation. This will adversely affect the computing budget, so take care not to use this option in that case.

PEXSI.NP-per-pole 4 *(integer)*

Number of MPI processes used to perform the PEXSI computations in one pole. If the total number of MPI processes is smaller than this number times the number of poles (times the spin multiplicity), the PEXSI library will compute appropriate groups of poles in sequence. The minimum time to solution is achieved by increasing this parameter as much as it is reasonable for parallel efficiency, and using enough MPI processes to allow complete parallelization over poles. On the other hand, the minimum computational cost (in the sense of computing budget) is obtained by using the minimum value of this parameter which is compatible with the memory footprint. The additional parallelization over poles will be irrelevant for cost, but it will obviously affect the time to solution.

Internally, SIESTA computes the processor grid parameters `nprow` and `npcol` for the PEXSI library, with `nprow` >= `npcol`, and as similar as possible. So it is best to choose **PEXSI.NP-per-pole** as the product of two similar numbers.

NOTE: The total number of MPI processes must be divisible by **PEXSI.NP-per-pole**. In case of spin-polarized calculations, the total number of MPI processes must be divisible by **PEXSI.NP-per-pole** times 2.

PEXSI.Ordering 1 *(integer)*

For large matrices, symbolic factorization should be performed in parallel to reduce the wall clock time. This can be done using ParMETIS/PT-Scotch by setting **PEXSI.Ordering** to 0. However, we have been experiencing some instability problem of the symbolic factorization phase when ParMETIS/PT-Scotch is used. In such case, for relatively small matrices one can either use the sequential METIS (**PEXSI.Ordering** = 1) or set **PEXSI.NP-symbfact** to 1.

PEXSI.NP-symbfact 1 *(integer)*

Number of MPI processes used to perform the symbolic factorizations needed in the PEXSI procedure. A default value should be given to reduce the instability problem. From experience so far setting this to be 1 is most stable, but going beyond 64 does not usually improve much.

PEXSI.Verbosity 1 *(integer)*

It determines the amount of information logged by the solver in different places. A value of zero gives minimal information.

- In the files logPEXSI[0-9]+, the verbosity level is interpreted by the PEXSI library itself. In the latest version, when PEXSI is compiled in RELEASE mode, only logPEXSI0 is given in the output. This is because we have observed that simultaneous output for all processors can have very significant cost for a large number of processors (>10000).
- In the SIESTA output file, a verbosity level of 1 and above will print lines (prefixed by `&o`) indicating the various heuristics used at each scf step. A verbosity level of 2 and above will print extra information.

The design of the output logging is still in flux.

6.13.3 Electron tolerance and the PEXSI solver

PEXSI.num-electron-tolerance 10^{-4} *(real)*

Tolerance in the number of electrons for the PEXSI solver. At each iteration of the solver, the number of electrons is computed as the trace of the density matrix times the overlap matrix, and compared with the total number of electrons in the system. This tolerance can be fixed, or dynamically determined as a function of the degree of convergence of the self-consistent-field loop.

PEXSI.num-electron-tolerance-lower-bound 10^{-2} *(real)*

See **PEXSI.num-electron-tolerance-upper-bound**.

PEXSI.num-electron-tolerance-upper-bound 0.5 *(real)*

The upper and lower bounds for the electron tolerance are used to dynamically change the tolerance in the PEXSI solver, following the simple algorithm:

```
tolerance = Max(lower_bound,Min(dDmax, upper_bound))
```

The first scf step uses the upper bound of the tolerance range, and subsequent steps use progressively lower values, in correspondence with the convergence-monitoring variable **dDmax**.

NOTE: This simple update schedule tends to work quite well. There is an experimental algorithm, documented only in the code itself, which allows a finer degree of control of the tolerance update.

PEXSI.mu-max-iter 10 *(integer)*

Maximum number of iterations of the PEXSI solver. Note that in this implementation there is no fallback procedure if the solver fails to converge in this number of iterations to the prescribed tolerance. In this case, the resulting density matrix might still be re-normalized, and the calculation able to continue, if the tolerance for non normalized DMs is not set too tight. For example,

```
# (true_no_electrons/no_electrons) - 1.0
  DM.NormalizationTolerance 1.0e-3
```

will allow a 0.1% error in the number of electrons. For obvious reasons, this feature, which is also useful in connection with the dynamic tolerance update, should not be abused.

If the parameters of the PEXSI solver are adjusted correctly (including a judicious use of inertia-counting to refine the μ bracket), we should expect that the maximum number of solver iterations needed is around 3

PEXSI.mu $-0.6\,\mathrm{Ry}$ *(energy)*

The starting guess for the chemical potential for the PEXSI solver. Note that this value does not affect the initial μ bracket for the inertia-count refinement, which is controlled by **PEXSI.mu-min** and **PEXSI.mu-max**. After an inertia-count phase, μ will be reset, and further iterations inherit this estimate, so this parameter is only relevant if there is no inertia-counting phase.

PEXSI.mu-pexsi-safeguard $0.05\,\mathrm{Ry}$ *(energy)*

NOTE: This feature has been deactivated for now. The condition for starting a new phase of

inertia-counting is that the Newton estimation falls outside the current bracket. The bracket is expanded accordingly.

The PEXSI solver uses Newton's method to update the estimate of μ. If the attempted change in μ is larger than **PEXSI.mu-pexsi-safeguard**, the solver cycle is stopped and a fresh phase of inertia-counting is started.

6.13.4 Inertia-counting

PEXSI.Inertia-Counts **3** *(integer)*

In a given scf step, the PEXSI procedure can optionally employ a μ bracket-refinement procedure based on inertia-counting. Typically, this is used only in the first few scf steps, and this parameter determines how many. If positive, inertia-counting will be performed for exactly that number of scf steps. If negative, inertia-counting will be performed for at least that number of scf steps, and then for as long as the scf cycle is not yet deemed to be near convergence (as determined by the **PEXSI.safe-dDmax-no-inertia** parameter).

NOTE: Since it is cheaper to perform an inertia-count phase than to execute one iteration of the solver, it pays to call the solver only when the μ bracket is sufficiently refined.

PEXSI.mu-min $-1\,\mathrm{Ry}$ *(energy)*

The lower bound of the initial range for μ used in the inertia-count refinement. In runs with multiple geometry iterations, it is used only for the very first scf iteration at the first geometry step. Further iterations inherit possibly refined values of this parameter.

PEXSI.mu-max $0\,\mathrm{Ry}$ *(energy)*

The upper bound of the initial range for μ used in the inertia-count refinement. In runs with multiple geometry iterations, it is used only for the very first scf iteration at the first geometry step. Further iterations inherit possibly refined values of this parameter.

PEXSI.safe-dDmax-no-inertia **0.05** *(real)*

During the scf cycle, the variable conventionally called **dDmax** monitors how far the cycle is from convergence. If **PEXSI.Inertia-Counts** is negative, an inertia-counting phase will be performed in a given scf step for as long as **dDmax** is greater than **PEXSI.safe-dDmax-no-inertia**.

NOTE: Even though **dDmax** represents historically how far from convergence the density-matrix is, the same mechanism applies to other forms of mixing in which other magnitudes are monitored for convergence (Hamiltonian, charge density...).

PEXSI.lateral-expansion-inertia $3\,\mathrm{eV}$ *(energy)*

If the correct μ is outside the bracket provided to the inertia-counting phase, the bracket is expanded in the appropriate direction(s) by this amoount.

PEXSI.Inertia-mu-tolerance $0.05\,\mathrm{Ry}$ *(energy)*

One of the criteria for early termination of the inertia-counting phase. The value of the estimated μ (basically the center of the resulting brackets) is monitored, and the cycle stopped if its change from one iteration to the next is below this parameter.

PEXSI.Inertia-max-iter 5 *(integer)*

Maximum number of inertia-count iterations per cycle.

PEXSI.Inertia-min-num-shifts 10 *(integer)*

Minimum number of sampling points for inertia counts.

PEXSI.Inertia-energy-width-tolerance ⟨**PEXSI.Inertia-mu-tolerance**⟩ *(energy)*

One of the criteria for early termination of the inertia-counting phase. The cycle stops if the width of the resulting bracket is below this parameter.

6.13.5 Re-use of μ information accross iterations

This is an important issue, as the efficiency of the PEXSI procedure depends on how close a guess of μ we have at our disposal. There are two types of information re-use:

- Bracketing information used in the inertia-counting phase.
- The values of μ itself for the solver.

PEXSI.safe-width-ic-bracket 4 eV *(energy)*

By default, the μ bracket used for the inertia-counting phase in scf steps other than the first is taken as an interval of width **PEXSI.safe-width-ic-bracket** around the latest estimate of μ.

PEXSI.safe-dDmax-ef-inertia 0.1 *(real)*

The change in μ from one scf iteration to the next can be crudely estimated by assuming that the change in the band structure energy (estimated as $\mathrm{Tr}\Delta H\mathrm{DM}$) is due to a rigid shift. When the scf cycle is near convergence, this $\Delta\mu$ can be used to estimate the new initial bracket for the inertia-counting phase, rigidly shifting the output bracket from the previous scf step. The cycle is assumed to be near convergence when the monitoring variable **dDmax** is smaller than **PEXSI.safe-dDmax-ef-inertia**.

NOTE: Even though **dDmax** represents historically how far from convergence the density-matrix is, the same mechanism applies to other forms of mixing in which other magnitudes are monitored for convergence (Hamiltonian, charge density...).

NOTE: This criterion will lead in general to tighter brackets than the previous one, but oscillations in H in the first few iterations might make it more dangerous. More information from real use cases is needed to refine the heuristics in this area.

PEXSI.safe-dDmax-ef-solver 0.05 *(real)*

When the scf cycle is near convergence, the $\Delta\mu$ estimated as above can be used to shift the initial guess for μ for the PEXSI solver. The cycle is assumed to be near convergence when the monitoring variable **dDmax** is smaller than **PEXSI.safe-dDmax-ef-solver**.

NOTE: Even though **dDmax** represents historically how far from convergence the density-matrix is, the same mechanism applies to other forms of mixing in which other magnitudes are monitored for convergence (Hamiltonian, charge density...).

PEXSI.safe-width-solver-bracket 4 eV *(energy)*

In all cases, a "safe" bracket around μ is provided even in direct calls to the PEXSI solver, in case a fallback to executing internally a cycle of inertia-counting is needed. The size of the bracket is given by **PEXSI.safe-width-solver-bracket**

6.13.6 Calculation of the density of states by inertia-counting

The cumulative or integrated density of states (INTDOS) can be easily obtained by inertia-counting, which involves a factorization of $H - \sigma S$ for varying σ (see SIESTA-PEXSI paper). Apart from the DOS-specific options below, the "ordering", "symbolic factorization", and "pole group size" (re-interpreted as the number of MPI processes dealing with a given σ) options are honored.

The current version of the code generates a file with the energy-INTDOS information, `PEXSI_INTDOS`, which can be later processed to generate the DOS by direct numerical differentiation, or a SIESTA-style `SystemLabel.EIG` file (using the `Util/PEXSI/intdos2eig` program).

PEXSI.DOS **false** *(logical)*

Whether to compute the DOS (actually, the INTDOS — see above) using the PEXSI technology.

PEXSI.DOS.Emin −1 Ry *(energy)*

Lower bound of energy window to compute the DOS in.

See **PEXSI.DOS.Ef.Reference**.

PEXSI.DOS.Emax 1 Ry *(energy)*

Upper bound of energy window to compute the DOS in.

See **PEXSI.DOS.Ef.Reference**.

PEXSI.DOS.Ef.Reference **true** *(logical)*

If this flag is true, the bounds of the energy window (**PEXSI.DOS.Emin** and **PEXSI.DOS.Emax**) are with respect to the Fermi level.

PEXSI.DOS.NPoints **200** *(integer)*

The number of points in the energy interval at which the DOS is computed. It is rounded up to the nearest multiple of the number of available factorization groups, as the operations are perfectly parallel and there will be no extra cost involved.

6.13.7 Calculation of the LDOS by selected-inversion

The local-density-of-states (LDOS) around a given reference energy ε, representing the contribution to the charge density of the states with eigenvalues in the vicinity of ε, can be obtained formally by a "one-pole expansion" with suitable broadening (see SIESTA-PEXSI paper).

Apart from the LDOS-specific options below, the "ordering", "verbosity", and "symbolic factorization" options are honored.

The current version of the code generates a real-space grid file with extension `SystemLabel.LDSI`, and (if netCDF is compiled-in) a file `Rho.grid.nc` (which unfortunately will overwrite any other charge-density files produced in the same run).

NOTE: The LDOS computed with this procedure is not exactly the same as the vanilla SIESTA LDOS, which uses an explicit energy interval. Here the broadening acts around a single value of the energy.

PEXSI.LDOS **false** *(logical)*

Whether to compute the LDOS using the PEXSI technology.

PEXSI.LDOS.Energy 0 Ry *(energy)*

The (absolute) energy at which to compute the LDOS.

PEXSI.LDOS.Broadening 0.01 Ry *(energy)*

The broadening parameter for the LDOS.

PEXSI.LDOS.NP-per-pole ⟨**PEXSI.NP-per-pole**⟩ *(integer)*

The value of this parameter supersedes **PEXSI.NP-per-pole** for the calculation of the LDOS, which otherwise would keep idle all but **PEXSI.NP-per-pole** MPI processes, as it essentially consists of a "one-pole" procedure.

6.14 Band-structure analysis

This calculation of the band structure is performed optionally after the geometry loop finishes, and the output information written to the `SystemLabel.bands` file (see below for the format).

BandLinesScale **pi/a** *(string)*

Specifies the scale of the k vectors given in **BandLines** and **BandPoints** below. The options are:

pi/a k-vector coordinates are given in Cartesian coordinates, in units of π/a, where a is the lattice constant

ReciprocalLatticeVectors k vectors are given in reciprocal-lattice-vector coordinates

NOTE: you might need to define explicitly a LatticeConstant tag in your fdf file if you do not already have one, and make it consistent with the scale of the k-points and any unit-cell vectors you might have already defined.

%block BandLines ⟨**None**⟩ *(block)*

Specifies the lines along which band energies are calculated (usually along high-symmetry directions). An example for an FCC lattice is:

```
%block BandLines
  1  1.000  1.000  1.000  L        # Begin at L
 20  0.000  0.000  0.000  \Gamma   # 20 points from L to gamma
 25  2.000  0.000  0.000  X        # 25 points from gamma to X
 30  2.000  2.000  2.000  \Gamma   # 30 points from X to gamma
%endblock BandLines
```

where the last column is an optional LaTeX label for use in the band plot. If only given points (not lines) are required, simply specify 1 in the first column of each line. The first column of the first line must be always 1.

NOTE: this block is not used if **BandPoints** is present.

%block BandPoints ⟨**None**⟩ *(block)*

Band energies are calculated for the list of arbitrary k points given in the block. Units defined by **BandLinesScale** as for **BandLines**. The generated `SystemLabel.bands` file will contain the k point coordinates (in a.u.) and the corresponding band energies (in eV). Example:

```
%block BandPoints
   0.000  0.000  0.000   # This is a comment. eg this is gamma
```

```
    1.000  0.000  0.000
    0.500  0.500  0.500
%endblock BandPoints
```

See also **BandLines**.

WriteKbands **false** *(logical)*

If **true**, it writes the coordinates of the $\vec{k}$ vectors defined for band plotting, to the main output file.

WriteBands **false** *(logical)*

If **true**, it writes the Hamiltonian eigenvalues corresponding to the $\vec{k}$ vectors defined for band plotting, in the main output file.

6.14.1 Format of the .bands file

```
FermiEnergy (all energies in eV) \\
kmin, kmax (along the k-lines path, i.e. range of k in the band plot) \\
Emin, Emax (range of all eigenvalues) \\
NumberOfBands, NumberOfSpins (1 or 2), NumberOfkPoints \\
k1, ((ek(iband,ispin,1),iband=1,NumberOfBands),ispin=1,NumberOfSpins) \\
k2, ek \\
 . \\
 . \\
 . \\
klast, ek \\
NumberOfkLines \\
kAtBegOfLine1, kPointLabel \\
kAtEndOfLine1, kPointLabel \\
  . \\
  . \\
  . \\
kAtEndOfLastLine, kPointLabel \\
```

The `gnubands` postprocessing utility program (found in the Util/Bands directory) reads the `SystemLabel.bands` for plotting. See the **BandLines** data descriptor above for more information.

6.14.2 Output of wavefunctions associated to bands

The user can optionally request that the wavefunctions corresponding to the computed bands be written to file. They are written to the `SystemLabel.bands.WFSX` file. The relevant options are:

WFS.Write.For.Bands **false** *(logical)*

Instructs the program to compute and write the wave functions associated to the bands specified (by a **BandLines** or a **BandPoints** block) to the file `SystemLabel.WFSX`.

The information in this file might be useful, among other things, to generate "fatbands" plots, in which both band eigenvalues and information about orbital projections is presented. See the `fat` program in the `Util/COOP` directory for details.

WFS.Band.Min **1** *(integer)*

Specifies the lowest band index of the wave-functions to be written to the file `SystemLabel.WFSX` for each k-point (all k-points in the band set are affected).

WFS.Band.Max **number of orbitals** *(integer)*

Specifies the highest band index of the wave-functions to be written to the file `SystemLabel.WFSX` for each k-point (all k-points in the band set are affected).

6.15 Output of selected wavefunctions

The user can optionally request that specific wavefunctions are written to file. These wavefunctions are re-computed after the geometry loop (if any) finishes, using the last (presumably converged) density matrix produced during the last self-consistent field loop (after a final mixing). They are written to the `SystemLabel.selected.WFSX` file.

Note that the complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to SystemLabel.fullBZ.WFSX if the **COOP.Write** option is in effect.

Note that the complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to a NetCDF file `WFS.nc` if the **Diag.UseNewDiagk** option is in effect.

WaveFuncKPointsScale **pi/a** *(string)*

Specifies the scale of the k vectors given in **WaveFuncKPoints** below. The options are:

pi/a k-vector coordinates are given in Cartesian coordinates, in units of π/a, where a is the lattice constant

ReciprocalLatticeVectors k vectors are given in reciprocal-lattice-vector coordinates

%block WaveFuncKPoints ⟨**None**⟩ *(block)*

Specifies the k-points at which the electronic wavefunction coefficients are written. An example for an FCC lattice is:

```
%block WaveFuncKPoints
0.000  0.000  0.000  from 1 to 10   # Gamma wavefuncs 1 to 10
2.000  0.000  0.000  1 3 5          # X wavefuncs 1,3 and 5
1.500  1.500  1.500                 # K wavefuncs, all
%endblock WaveFuncKPoints
```

The index of a wavefunction is defined by its energy, so that the first one has lowest energy.

The user can also narrow the energy-range used with the **WFS.Energy.Min** and **WFS.Energy.Max** options (both take an energy (with units) as extra argument – see section 6.17.3). Care should be taken to make sure that the actual values of the options make sense.

The output of the wavefunctions in described in Section 6.15.

WriteWaveFunctions **false** *(logical)*

If **true**, it writes to the output file a list of the wavefunctions actually written to the `SystemLabel.selected.WFSX` file, which is always produced.

The unformatted WFSX file contains the information of the k-points for which wavefunctions coefficients are written, and the energies and coefficients of each wavefunction which was specified in the input file (see **WaveFuncKPoints** descriptor above). It also contains information on the atomic species and the orbitals for postprocessing purposes.

NOTE: The `SystemLabel.WFSX` file is in a more compact form than the old WFS, and the wavefunctions are output in single precision. The `Util/WFS/wfsx2wfs` program can be used to convert to the old format.

The `readwf` and `readwfsx` postprocessing utilities programs (found in the Util/WFS directory) read the `SystemLabel.WFS` or `SystemLabel.WFSX` files, respectively, and generate a readable file.

6.16 Densities of states

6.16.1 Total density of states

There are several options to obtain the total density of states:

- The Hamiltonian eigenvalues for the SCF sampling $\vec{k}$ points can be dumped into SystemLabel.EIG in a format analogous to SystemLabel.bands, but without the kmin, kmax, emin, emax information, and without the abscissa. The `Eig2DOS` postprocessing utility can be then used to obtain the density of states. See the **WriteEigenvalues** descriptor.
- As a side-product of a partial-density-of-states calculation (see below)
- As one of the files produced by the `Util/COOP/mprop` during the off-line analysis of the electronic structure. This method allows the flexibility of specifying energy ranges and resolutions at will, without re-running SIESTA See Sec. 6.17.3.
- Using the inertia-counting routines in the PEXSI solver (see Sec. 6.13.6).

6.16.2 Partial (projected) density of states

There are two options to obtain the partial density of states

- Using the options below
- Using the `Util/COOP/mprop` program for the off-line analysis of the electronic structure in PDOS mode. This method allows the flexibility of specifying energy ranges, orbitals, and resolutions at will, without re-running SIESTA. See Sec. 6.17.3.

%block ProjectedDensityOfStates ⟨**None**⟩ *(block)*

Instructs to write the Total Density Of States (Total DOS) and the Projected Density Of States (PDOS) on the basis orbitals, between two given energies, in files `SystemLabel.DOS` and `SystemLabel.PDOS`, respectively. The block must be a single line with the energies of the range for PDOS projection, (relative to the program's zero, i.e. the same as the eigenvalues printed by the program), the peak width (an energy) for broadening the eigenvalues, the number of points in the energy window, and the energy units. An example is:

```
%block ProjectedDensityOfStates
   -20.00  10.00  0.200  500  eV
%endblock ProjectedDensityOfStates
```

By default the projected density of states is generated for the same grid of points in reciprocal space as used for the SCF calculation. However, a separate set of K-points, usually on a finer grid, can be generated using one of the options **PDOS.kgrid.Cutoff** or **PDOS.kgrid.MonkhorstPack**. The format of these options is exactly the same as for **kgrid.Cutoff** and **kgrid.MonkhorstPack**, respectively. Note that if a gamma point calculation is being used in the SCF part, especially as part of a geometry optimisation, and this is then to be run with a grid of K-points for the PDOS calculation it is more efficient to run the SCF phase first and then restart to perform the PDOS evaluation using the density matrix saved from the SCF phase.

NOTE: the two energies of the range must be ordered, with lowest first.

The total DOS is stored in a file called `SystemLabel.DOS`. The format of this file is:

```
Energy value, Total DOS (spin up), Total DOS (spin down)
```

The Projected Density Of States for all the orbitals in the unit cell is dumped sequentially into a file called `SystemLabel.PDOS`. This file is structured using spacing and xml tags. A machine-readable (but not very human readable) xml file `pdos.xml` is also produced. Both can be processed by the program in `Util/pdosxml`. The `SystemLabel.PDOS` file can be processed by utilites in `Util/Contrib/APostnikov`.

In all cases, the units for the DOS are (number of states/eV), and the Total DOS, $g(\epsilon)$, is normalized as follows:

$$\int_{-\infty}^{\infty} g(\epsilon) d\epsilon = \text{number of basis orbitals in unit cell} \tag{14}$$

6.16.3 Local density of states

The LDOS is formally the DOS weighted by the amplitude of the corresponding wavefunctions at different points in space, and is then a function of energy and position. SIESTA can output the LDOS integrated over a range of energies. This information can be used to obtain simple STM images in the Tersoff-Hamann approximation (See `Util/STM/simple-stm`).

%block LocalDensityOfStates ⟨**None**⟩ *(block)*

Instructs to write the LDOS, integrated between two given energies, at the mesh used by DHSCF, in file `SystemLabel.LDOS`. This file can be read by routine IORHO, which may be used by an application program in later versions. The block must be a single line with the energies of the range for LDOS integration (relative to the program's zero, i.e. the same as the eigenvalues printed by the program) and their units. An example is:

```
%block LocalDensityOfStates
   -3.50  0.00   eV
%endblock LocalDensityOfStates
```

NOTE: the two energies of the range must be ordered, with lowest first.

6.17 Options for chemical analysis

6.17.1 Mulliken charges and overlap populations

WriteMullikenPop **0** *(integer)*

It determines the level of Mulliken population analysis printed:

0 none

1 atomic and orbital charges

2 atomic, orbital and atomic overlap populations

3 atomic, orbital, atomic overlap and orbital overlap populations

The order of the orbitals in the population lists is defined by the order of atoms. For each atom, populations for PAO orbitals and double-z, triple-z, etc... derived from them are displayed first for all the angular momenta. Then, populations for perturbative polarization orbitals are written. Within a l-shell be aware that the order is not conventional, being y, z, x for p orbitals, and xy, yz, z^2, xz, and $x^2 - y^2$ for d orbitals.

MullikenInSCF **false** *(logical)*

If **true**, the Mulliken populations will be written for every SCF step at the level of detail specified in **WriteMullikenPop**. Useful when dealing with SCF problems, otherwise too verbose.

6.17.2 Voronoi and Hirshfeld atomic population analysis

WriteHirshfeldPop **false** *(logical)*

If **true**, the program calculates and prints the Hirshfeld "net" atomic populations on each atom in the system. For a definition of the Hirshfeld charges, see Hirshfeld, Theo Chem Acta **44**, 129 (1977) and Fonseca et al, J. Comp. Chem. **25**, 189 (2003). Hirshfeld charges are more reliable than Mulliken charges, specially for large basis sets. The number printed is the total net charge of the atom: the variation from the neutral charge, in units of $|e|$: positive (negative) values indicate deficiency (excess) of electrons in the atom.

WriteVoronoiPop **false** *(logical)*

If **true**, the program calculates and prints the Voronoi "net" atomic populations on each atom in the system. For a definition of the Voronoi charges, see Bickelhaupt et al, Organometallics **15**, 2923 (1996) and Fonseca et al, J. Comp. Chem. **25**, 189 (2003). Voronoi charges are more reliable than Mulliken charges, specially for large basis sets. The number printed is the total net charge of the atom: the variation from the neutral charge, in units of $|e|$: positive (negative) values indicate deficiency (excess) of electrons in the atom.

The Hirshfeld and Voronoi populations (partial charges) are computed by default only at the end of the program (i.e., for the final geometry, after self-consistency). The following options allow more control:

PartialChargesAtEveryGeometry **false** *(logical)*

The Hirshfeld and Voronoi populations are computed after self-consistency is achieved, for all the geometry steps.

PartialChargesAtEverySCFStep **false** *(logical)*

The Hirshfeld and Voronoi populations are computed for every step of the self-consistency process.

Performance note: The default behavior (computing at the end of the program) involves an extra calculation of the charge density.

6.17.3 Crystal-Orbital overlap and hamilton populations (COOP/COHP)

These curves are quite useful to analyze the electronic structure to get insight about bonding characteristics. See the `Util/COOP` directory for more details. The **COOP.Write** option must be activated to get the information needed.

References:

- Original COOP reference: Hughbanks, T.; Hoffmann, R., J. Am. Chem. Soc., 1983, 105, 3528.
- Original COHP reference: Dronskowski, R.; BlÃűchl, P. E., J. Phys. Chem., 1993, 97, 8617.
- A tutorial introduction: Dronskowski, R. Computational Chemistry of Solid State Materials; Wiley-VCH: Weinheim, 2005.
- Online material maintained by R. Dronskowski's group: `http://www.cohp.de/`

COOP.Write **false** *(logical)*

Instructs the program to generate `SystemLabel.fullBZ.WFSX` (packed wavefunction file) and `SystemLabel.HSX` (H, S and X_ ij file), to be processed by `Util/COOP/mprop` to generate COOP/COHP curves, (projected) densities of states, etc.

The `.WFSX` file is in a more compact form than the usual `.WFS`, and the wavefunctions are output in single precision. The `Util/wfsx2wfs` program can be used to convert to the old format. The HSX file is in a more compact form than the usual HS, and the Hamiltonian, overlap matrix, and relative-positions array (which is always output, even for gamma-point only calculations) are in single precision.

The user can narrow the energy-range used (and save some file space) by using the **WFS.Energy.Min** and **WFS.Energy.Max** options (both take an energy (with units) as extra argument), and/or the **WFS.Band.Min** and **WFS.Band.Max** options. Care should be taken to make sure that the actual values of the options make sense.

Note that the band range options could also affect the output of wave-functions associated to bands (see section 6.14.2), and that the energy range options could also affect the output of user-selected wave-functions with the **WaveFuncKPoints** block (see section 6.15).

WFS.Energy.Min $-\infty$ *(energy)*

Specifies the lowest value of the energy (eigenvalue) of the wave-functions to be written to the file `SystemLabel.fullBZ.WFSX` for each k-point (all k-points in the BZ sampling are affected).

WFS.Energy.Max ∞ *(energy)*

Specifies the highest value of the energy (eigenvalue) of the wave-functions to be written to the file `SystemLabel.fullBZ.WFSX` for each k-point (all k-points in the BZ sampling are affected).

6.18 Optical properties

OpticalCalculation **false** *(logical)*

If specified, the imaginary part of the dielectric function will be calculated and stored in a file called `SystemLabel.EPSIMG`. The calculation is performed using the simplest approach based on the dipolar transition matrix elements between different eigenfunctions of the self-consistent Hamiltonian. For molecules the calculation is performed using the position operator matrix elements, while for solids the calculation is carried out in the momentum space formulation. Corrections due to the non-locality of the pseudopotentials are introduced in the usual way.

Optical.Energy.Minimum 0 Ry *(energy)*

This specifies the minimum of the energy range in which the frequency spectrum will be calculated.

Optical.Energy.Maximum 10 Ry *(energy)*

This specifies the maximum of the energy range in which the frequency spectrum will be calculated.

Optical.Broaden 0 Ry *(energy)*

If this is value is set then a Gaussian broadening will be applied to the frequency values.

Optical.Scissor 0 Ry *(energy)*

Because of the tendency of DFT calculations to under estimate the band gap, a rigid shift of the unoccupied states, known as the scissor operator, can be added to correct the gap and thereby improve the calculated results. This shift is only applied to the optical calculation and no where else within the calculation.

Optical.NumberOfBands **all bands** *(integer)*

This option controls the number of bands that are included in the optical property calculation. Clearly this number must be larger than the number of occupied bands and less than or equal to the number of basis functions (which determines the number of unoccupied bands available). Note, while including all the bands may be the most accurate choice this will also be the most expensive!

%block Optical.Mesh ⟨**None**⟩ *(block)*

This block contains 3 numbers that determine the mesh size used for the integration across the Brillouin zone. For example:

```
%block  Optical.Mesh
  5 5 5
%endblock  Optical.Mesh
```

The three values represent the number of mesh points in the direction of each reciprocal lattice vector.

Optical.OffsetMesh **false** *(logical)*

If set to true, then the mesh is offset away from the gamma point for odd numbers of points.

Optical.PolarizationType **polycrystal** *(string)*

This option has three possible values that represent the type of polarization to be used in the

calculation. The options are

polarized implies the application of an electric field in a given direction

unpolarized implies the propagation of light in a given direction

polycrystal In the case of the first two options a direction in space must be specified for the electric field or propagation using the **Optical.Vector** data block.

%block Optical.Vector ⟨**None**⟩ *(block)*

This block contains 3 numbers that specify the vector direction for either the electric field or light propagation, for a polarized or unpolarized calculation, respectively. A typical block might look like:

```
%block  Optical.Vector
  1.0 0.0 0.5
%endblock  Optical.Vector
```

6.19 Macroscopic polarization

%block PolarizationGrids ⟨**None**⟩ *(block)*

If specified, the macroscopic polarization will be calculated using the geometric Berry phase approach (R.D. King-Smith, and D. Vanderbilt, PRB **47**, 1651 (1993)). In this method the electronic contribution to the macroscopic polarization, along a given direction, is calculated using a discretized version of the formula

$$P_{e,\parallel} = \frac{if q_e}{8\pi^3} \int_A d\mathbf{k}_\perp \sum_{n=1}^{M} \int_0^{|G_\parallel|} dk_\parallel \langle u_{\mathbf{k}n} | \frac{\delta}{\delta k_\parallel} | u_{\mathbf{k}n} \rangle \tag{15}$$

where f is the occupation (2 for a non-magnetic system), q_e the electron charge, M is the number of occupied bands (the system **must** be an insulator), and $u_{\mathbf{k}n}$ are the periodic Bloch functions. $\mathbf{G}_\parallel$ is the shortest reciprocal vector along the chosen direction.

As it can be seen in formula (15), to compute each component of the polarization we must perform a surface integration of the result of a 1-D integral in the selected direction. The grids for the calculation along the direction of each of the three lattice vectors are specified in the block **PolarizationGrids**.

```
%block PolarizationGrids
   10   3  4      yes
    2  20  2       no
    4   4 15
%endblock PolarizationGrids
```

All three grids must be specified, therefore a 3×3 matrix of integer numbers must be given: the first row specifies the grid that will be used to calculate the polarization along the direction of the first lattice vector, the second row will be used for the calculation along the the direction of the second lattice vector, and the third row for the third lattice vector. The numbers in the diagonal of the matrix specifie the number of points to be used in the one dimensional line integrals along the different directions. The other numbers specifie the mesh used in the surface integrals. The last column specifies if the bidimensional grids are going to be diplaced from the origin or not, as in the Monkhorst-Pack algorithm (PRB **13**, 5188 (1976)). This last

column is optional. If the number of points in one of the grids is zero, the calculation will not be performed for this particular direction.

For example, in the given example, for the computation in the direction of the first lattice vector, 15 points will be used for the line integrals, while a 3×4 mesh will be used for the surface integration. This last grid will be displaced from the origin, so Γ will not be included in the bidimensional integral. For the directions of the second and third lattice vectors, the number of points will be 20 and 2×2, and 15 and 4×4, respectively.

It has to be stressed that the macroscopic polarization can only be meaningfully calculated using this approach for insulators. Therefore, the presence of an energy gap is necessary, and no band can cross the Fermi level. The program performs a simple check of this condition, just by counting the electrons in the unit cell (the number must be even for a non-magnetic system, and the total spin polarization must have an integer value for spin polarized systems), however is the responsability of the user to check that the system under study is actually an insulator (for both spin components if spin polarized).

The total macroscopic polarization, given in the output of the program, is the sum of the electronic contribution (calculated as the Berry phase of the valence bands), and the ionic contribution, which is simply defined as the sum of the atomic positions within the unit cell multiply by the ionic charges ($\sum_i^{N_a} Z_i \mathbf{r}_i$). In the case of the magnetic systems, the bulk polarization for each spin component has been defined as

$$\mathbf{P}^\sigma = \mathbf{P}_e^\sigma + \frac{1}{2} \sum_i^{N_a} Z_i \mathbf{r}_i \tag{16}$$

N_a is the number of atoms in the unit cell, and $\mathbf{r}_i$ and Z_i are the positions and charges of the ions.

It is also worth noting, that the macroscopic polarization given by formula (15) is only defined modulo a "quantum" of polarization (the bulk polarization per unit cell is only well defined modulo $f q_e \mathbf{R}$, being $\mathbf{R}$ an arbitrary lattice vector). However, the experimentally observable quantities are associated to changes in the polarization induced by changes on the atomic positions (dynamical charges), strains (piezoelectric tensor), etc... The calculation of those changes, between different configurations of the solid, will be well defined as long as they are smaller than the "quantum", i.e. the perturbations are small enough to create small changes in the polarization.

BornCharge false *(logical)*

If true, the Born effective charge tensor is calculated for each atom by finite differences, by calculating the change in electric polarization (see **PolarizationGrids**) induced by the small displacements generated for the force constants calculation (see **MD.TypeOfRun FC**):

$$Z^*_{i,\alpha,\beta} = \frac{\Omega_0}{e} \left. \frac{\partial P_\alpha}{\partial u_{i,\beta}} \right|_{q=0} \tag{17}$$

where e is the charge of an electron and Ω_0 is the unit cell volume.

To calculate the Born charges it is necessary to specify both the Born charge flag and the mesh used to calculate the polarization, for example:

```
%block PolarizationGrids
  7  3  3
```

```
  3  7  3
  3  3  7
%endblock PolarizationGrids
BornCharge True
```

The Born effective charge matrix is then written to the file `SystemLabel.BC`.

The method by which the polarization is calculated may introduce an arbitrary phase (polarization quantum), which in general is far larger than the change in polarization which results from the atomic displacement. It is removed during the calculation of the Born effective charge tensor.

The Born effective charges allow the calculation of LO-TO splittings and infrared activities. The version of the Vibra utility code in which these magnitudes are calculated is not yet distributed with SIESTA, but can be obtained form Tom Archer (archert@tcd.ie).

6.20 Maximally Localized Wannier Functions. Interface with the wannier90 code

`wannier90` (http://www.wannier.org) is a code to generate maximally localized wannier functions according to the original Marzari and Vanderbilt recipe.

It is strongly recommended to read the original papers on which this method is based and the documentation of `wannier90` code. Here we shall focus only on those internal SIESTA variables required to produce the files that will be processed by `wannier90`.

A complete list of examples and tests (including molecules, metals, semiconductors, insulators, magnetic systems, plotting of Fermi surfaces or interpolation of bands), can be downloaded from

http://personales.unican.es/junqueraj/Wannier-examples.tar.gz

NOTE: The Bloch functions produced by a first-principles code have arbitrary phases that depend on the number of processors used and other possibly non-reproducible details of the calculation. In what follows it is essential to maintain consistency in the handling of the overlap and Bloch-funcion files produced and fed to `wannier90`.

Siesta2Wannier90.WriteMmn **false** *(logical)*

This flag determines whether the overlaps between the periodic part of the Bloch states at neighbour k-points are computed and dumped into a file in the format required by `wannier90`. These overlaps are defined in Eq. (27) in the paper by N. Marzari *et al.*, Review of Modern Physics **84**, 1419 (2012), or Eq. (1.7) of the Wannier90 User Guide, Version 2.0.1.

The k-points for which the overlaps will be computed are read from a `.nnkp` file produced by `wannier90`. It is strongly recommended for the user to read the corresponding user guide.

The overlap matrices are written in a file with extension `.mmn`.

Siesta2Wannier90.WriteAmn **false** *(logical)*

This flag determines whether the overlaps between Bloch states and trial localized orbitals are computed and dumped into a file in the format required by `wannier90`. These projections are defined in Eq. (16) in the paper by N. Marzari *et al.*, Review of Modern Physics **84**, 1419 (2012), or Eq. (1.8) of the Wannier90 User Guide, Version 2.0.1.

The localized trial functions to use are taken from the `.nnkp` file produced by `wannier90`. It is strongly recommended for the user to read the corresponding user guide.

The overlap matrices are written in a file with extension `.amn`.

Siesta2Wannier90.WriteEig **false** *(logical)*

Flag that determines whether the Kohn-Sham eigenvalues (in eV) at each point in the Monkhorst-Pack mesh required by `wannier90` are written to file. This file is mandatory in `wannier90` if any of disentanglement, plot_bands, plot_fermi_surface or hr_plot options are set to true in the `wannier90` input file.

The eigenvalues are written in a file with extension `.eigW`. This extension is chosen to avoid name clashes with SIESTA's standard eigenvalue file in case-insensitive filesystems.

Siesta2Wannier90.WriteUnk **false** *(logical)*

Produces `UNKXXXXX.Y` files which contain the periodic part of a Bloch function in the unit cell on a grid given by global unk_nx, unk_ny, unk_nz variables. The name of the output files is assumed to have the previous form, where the `XXXXX` refer to the k-point index (from 00001 to the total number of k-points considered), and the `Y` refers to the spin component (1 or 2)

The periodic part of the Bloch functions is defined by

$$u_{n\vec{k}}(\vec{r}) = \sum_{\vec{R}\mu} c_{n\mu}(\vec{k}) e^{i\vec{k}\cdot(\vec{r}_\mu + \vec{R} - \vec{r})} \phi_\mu(\vec{r} - \vec{r}_\mu - \vec{R}), \tag{18}$$

where $\phi_\mu(\vec{r} - \vec{r}_\mu - \vec{R})$ is a basis set atomic orbital centered on atom μ in the unit cell $\vec{R}$, and $c_{n\mu}(\vec{k})$ are the coefficients of the wave function. The latter must be identical to the ones used for wannierization in M_{mn}. (See the above comment about arbitrary phases.)

Siesta2Wannier90.UnkGrid1 **⟨mesh points along A⟩** *(integer)*

Number of points along the first lattice vector in the grid where the periodic part of the wave functions will be plotted.

Siesta2Wannier90.UnkGrid2 **⟨mesh points along B⟩** *(integer)*

Number of points along the second lattice vector in the grid where the periodic part of the wave functions will be plotted.

Siesta2Wannier90.UnkGrid3 **⟨mesh points along C⟩** *(integer)*

Number of points along the third lattice vector in the grid where the periodic part of the wave functions will be plotted.

Siesta2Wannier90.UnkGridBinary **true** *(logical)*

Flag that determines whether the periodic part of the wave function in the real space grid is written in binary format (default) or in ASCII format.

Siesta2Wannier90.NumberOfBands **occupied bands** *(integer)*

In spin unpolarized calculations, number of bands that will be initially considered by SIESTA to generate the information required by `wannier90`. Note that it should be at least as large as the index of the highest-lying band in the `wannier90` post-processing. For example, if the wannierization is going to involve bands 3 to 5, the SIESTA number of bands should be at least 5. Bands 1 and 2 should appear in a "excluded" list.

NOTE: you are highly encouraged to explicitly specify the number of bands.

Siesta2Wannier90.NumberOfBandsUp **⟨Siesta2Wannier90.NumberOfBands⟩** *(integer)*

In spin-polarized calculations, number of bands with spin up that will be initially considered by SIESTA to generate the information required by `wannier90`.

Siesta2Wannier90.NumberOfBandsDown ⟨**Siesta2Wannier90.NumberOfBands**⟩
(integer)

In spin-polarized calculations, number of bands with spin down that will be initially considered by SIESTA to generate the information required by `wannier90`.

6.21 Systems with net charge or dipole, and electric fields

NetCharge 0 *(real)*

Specify the net charge of the system (in units of $|e|$). For charged systems, the energy converges very slowly versus cell size. For molecules or atoms, a Madelung correction term is applied to the energy to make it converge much faster with cell size (this is done only if the cell is SC, FCC or BCC). For other cells, or for periodic systems (chains, slabs or bulk), this energy correction term can not be applied, and the user is warned by the program. It is not advised to do charged systems other than atoms and molecules in SC, FCC or BCC cells, unless you know what you are doing.

Use: For example, the F^- ion would have **NetCharge -1** , and the Na^+ ion would have **NetCharge 1**. Fractional charges can also be used.

SimulateDoping **false** *(logical)*

This option instructs the program to add a background charge density to simulate doping. The new "doping" routine calculates the net charge of the system, and adds a compensating background charge that makes the system neutral. This background charge is constant at points of the mesh near the atoms, and zero at points far from the atoms. This simulates situations like doped slabs, where the extra electrons (holes) are compensated by oposite charges at the material (the ionized dopant impurities), but not at the vacuum. This serves to simulate properly doped systems in which there are large portions of vacuum, such as doped slabs.

(See `Tests/sic-slab`)

%block ExternalElectricField ⟨**None**⟩ *(block)*

It specifies an external electric field for molecules, chains and slabs. The electric field should be orthogonal to 'bulk directions', like those parallel to a slab (bulk electric fields, like in dielectrics or ferroelectrics, are not allowed). If it is not, an error message is issued and the components of the field in bulk directions are suppressed automatically. The input is a vector in Cartesian coordinates, in the specified units. Example:

```
%block ExternalElectricField
   0.000  0.000  0.500  V/Ang
%endblock ExternalElectricField
```

Starting with version 4.0, applying an electric field perpendicular to a slab will by default enable the slab dipole correction, see **SlabDipoleCorrection**. To reproduce older calculations, set this correction option explicitly to **false** in the input file.

SlabDipoleCorrection **false** *(logical)*

If **true**, SIESTA calculates the electric field required to compensate the dipole of the system at every iteration of the self-consistent cycle. The potential added to the grid corresponds to

that of a dipole layer at the middle of the vacuum layer. For slabs, this exactly compensates the electric field at the vacuum created by the dipole moment of the system, thus allowing to treat asymmetric slabs (including systems with an adsorbate on one surface) and compute properties such as the work funcion of each of the surfaces.

NOTE: If the program is fed a starting density matrix from an uncorrected calculation (i.e., with an exagerated dipole), the first iteration might use a compensating field that is too big, with the risk of taking the system out of the convergence basin. In that case, it is advisable to use the **SCF.Mix.First** option to request a mix of the input and output density matrices after that first iteration.

(See `Tests/sic-slab`)

This will default to **true** if an external field is applied to a slab calculation, otherwise it will default to **false**.

%block Geometry.Hartree ⟨**None**⟩ *(block)*

Allow introduction of regions with changed Hartree potential. Introducing a potential can act as a repulsion (positive value) or attraction (negative value) region.

The regions are defined as geometrical objects and there are no limits to the number of defined geometries.

Currently 4 different kinds of geometries are allowed:

Infinite plane Define a geometry by an infinite plane which cuts the unit-cell.

This geometry is defined by a single point which is in the plane and a vector normal to the plane.

This geometry has 3 different settings:

delta An infinite plane with δ-height.

gauss An infinite plane with a Gaussian distributed height profile.

exp An infinite plane with an exponentially distributed height profile.

Bounded plane Define a geometric plane which is bounded, i.e. not infinite.

This geometry is defined by an origo of the bounded plane and two vectors which span the plane, both originating in the respective origo.

This geometry has 3 different settings:

delta A plane with δ-height.

gauss A plane with a Gaussian distributed height profile.

exp A plane with an exponentially distributed height profile.

Box This geometry is defined by an origo of the box and three vectors which span the box, all originating from the respective origo.

This geometry has 1 setting:

delta No decay-region outside the box.

Spheres This geometry is defined by a list of spheres and a common radii.

This geometry has 2 settings:

gauss All spheres have an gaussian distribution about their centre.

exp All spheres have an exponential decay.

Here is a list of all options combined in one block:

```
%block Geometry.Hartree
 plane   1. eV       # The lifting potential on the geometry
   delta
    1.0 1.0 1.0 Ang  # An intersection point, in the plane
    1.0 0.5 0.2      # The normal vector to the plane
 plane  -1. eV       # The lifting potential on the geometry
   gauss 1. 2.  Ang  # the std. and the cut-off length
    1.0 1.0 1.0 Ang  # An intersection point, in the plane
    1.0 0.5 0.2      # The normal vector to the plane
 plane   1. eV       # The lifting potential on the geometry
   exp 1. 2. Ang     # the half-length and the cut-off length
    1.0 1.0 1.0 Ang  # An intersection point, in the plane
    1.0 0.5 0.2      # The normal vector to the plane
 square  1. eV       # The lifting potential on the geometry
   delta
    1.0 1.0 1.0 Ang  # The starting point of the square
    2.0 0.5 0.2 Ang  # The first spanning vector
    0.0 2.5 0.2 Ang  # The second spanning vector
 square  1. eV       # The lifting potential on the geometry
   gauss 1. 2. Ang   # the std. and the cut-off length
    1.0 1.0 1.0 Ang  # The starting point of the square
    2.0 0.5 0.2 Ang  # The first spanning vector
    0.0 2.5 0.2 Ang  # The second spanning vector
 square  1. eV       # The lifting potential on the geometry
   exp 1. 2. Ang     # the half-length and the cut-off length
    1.0 1.0 1.0 Ang  # The starting point of the square
    2.0 0.5 0.2 Ang  # The first spanning vector
    0.0 2.5 0.2 Ang  # The second spanning vector
 box  1. eV          # The lifting potential on the geometry
   delta
    1.0 1.0 1.0 Ang  # Origo of the box
    2.0 0.5 0.2 Ang  # The first spanning vector
    0.0 2.5 0.2 Ang  # The second spanning vector
    0.0 0.5 3.2 Ang  # The third spanning vector
 coords 1. eV        # The lifting potential on the geometry
    gauss 2. 4. Ang  # First is std. deviation, second is cut-off radii
       2 spheres     # How many spheres in the following lines
       0.0 4. 2. Ang # The centre coordinate of 1. sphere
       1.3 4. 2. Ang # The centre coordinate of 2. sphere
 coords 1. eV        # The lifting potential on the geometry
    exp 2. 4. Ang    # First is half-length, second is cut-off radii
       2 spheres     # How many spheres in the following lines
       0.0 4. 2. Ang # The centre coordinate of 1. sphere
       1.3 4. 2. Ang # The centre coordinate of 2. sphere
%endblock Geometry.Hartree
```

%block Geometry.Charge ⟨None⟩ *(block)*

This is similar to the **Geometry.Hartree** block. However, instead of specifying a potential, one defines the total charge that is spread on the geometry.

To see how the input should be formatted, see **Geometry.Hartree** and remove the unit-specification. Note that the input value is number of electrons (similar to **NetCharge**).

6.22 Output of charge densities and potentials on the grid

SIESTA represents these magnitudes on the real-space grid. The following options control the generation of the appropriate files, which can be processed by the programs in the `Util/Grid` directory, and also by Andrei Postnikov's utilities in `Util/Contrib/APostnikov`. See also `Util/Denchar` for an alternative way to plot the charge density (and wavefunctions).

SaveRho **false** *(logical)*

Instructs to write the valence pseudocharge density at the mesh used by DHSCF, in file `SystemLabel.RHO`.

NOTE: file `.RHO` is only written, not read, by siesta. This file can be read by routine IORHO, which may be used by other application programs.

If netCDF support is compiled in, the file `Rho.grid.nc` is produced.

SaveDeltaRho **false** *(logical)*

Instructs to write $\delta\rho(\vec{r}) = \rho(\vec{r}) - \rho_{atm}(\vec{r})$, i.e., the valence pseudocharge density minus the sum of atomic valence pseudocharge densities. It is done for the mesh points used by DHSCF and it comes in file `SystemLabel.DRHO`. This file can be read by routine IORHO, which may be used by an application program in later versions.

NOTE: file `.DRHO` is only written, not read, by siesta.

If netCDF support is compiled in, the file `DeltaRho.grid.nc` is produced.

SaveRhoXC **false** *(logical)*

Instructs to write the valence pseudocharge density at the mesh, including the nonlocal core corrections used to calculate the exchange-correlation energy, in file `SystemLabel.RHOXC`.

Use: File `.RHOXC` is only written, not read, by siesta.

If netCDF support is compiled in, the file `RhoXC.grid.nc` is produced.

SaveElectrostaticPotential **false** *(logical)*

Instructs to write the total electrostatic potential, defined as the sum of the hartree potential plus the local pseudopotential, at the mesh used by DHSCF, in file `SystemLabel.VH`. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `.VH` is only written, not read, by siesta.

If netCDF support is compiled in, the file `ElectrostaticPotential.grid.nc` is produced.

SaveNeutralAtomPotential **false** *(logical)*

Instructs to write the neutral-atom potential, defined as the sum of the hartree potential of a "pseudo atomic valence charge" plus the local pseudopotential, at the mesh used by DHSCF, in file `SystemLabel.VNA`. It is written at the start of the self-consistency cycle, as this potential does not change.

Use: File `.VNA` is only written, not read, by siesta.

If netCDF support is compiled in, the file `Vna.grid.nc` is produced.

SaveTotalPotential **false** *(logical)*

Instructs to write the valence total effective local potential (local pseudopotential + Hartree + Vxc), at the mesh used by DHSCF, in file `SystemLabel.VT`. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `.VT` is only written, not read, by siesta.

If netCDF support is compiled in, the file `TotalPotential.grid.nc` is produced.

NOTE: a side effect; the vacuum level, defined as the effective potential at grid points with zero density, is printed in the standard output whenever such points exist (molecules, slabs) and either **SaveElectrostaticPotential** or **SaveTotalPotential** are **true**. In a symetric (nonpolar) slab, the work function can be computed as the difference between the vacuum level and the Fermi energy.

SaveIonicCharge **false** *(logical)*

Instructs to write the soft diffuse ionic charge at the mesh used by DHSCF, in file `SystemLabel.IOCH`. This file can be read by routine IORHO, which may be used by an application program in later versions. Remember that, within the SIESTA sign convention, the electron charge density is positive and the ionic charge density is negative.

Use: File `.IOCH` is only written, not read, by siesta.

If netCDF support is compiled in, the file `Chlocal.grid.nc` is produced.

SaveTotalCharge **false** *(logical)*

Instructs to write the total charge density (ionic+electronic) at the mesh used by DHSCF, in file `SystemLabel.TOCH`. This file can be read by routine IORHO, which may be used by an application program in later versions. Remember that, within the SIESTA sign convention, the electron charge density is positive and the ionic charge density is negative.

Use: File `.TOCH` is only written, not read, by siesta.

SaveBaderCharge **false** *(logical)*

Instructs the program to save the charge density for further post-processing by a Bader-analysis program. This "Bader charge" is the sum of the electronic valence charge density and a set of "model core charges" placed at the atomic sites. For a given atom, the model core charge is a generalized Gaussian, but confined to a radius of 1.0 Bohr (by default), and integrating to the total core charge (Z-$Z_{\rm val}$). These core charges are needed to provide local maxima for the charge density at the atomic sites, which are not guaranteed in a pseudopotential calculation. For hydrogen, an artificial core of 1 electron is added, with a confinement radius of 0.6 Bohr by default. The Bader charge is projected on the grid points of the mesh used by DHSCF, and saved in file `SystemLabel.BADER`. This file can be post-processed by the program `Util/grid2cube` to convert it to the "cube" format, accepted by several Bader-analysis programs (for example, see `http://theory.cm.utexas.edu/bader/`). Due to the need to represent a localized core charge, it is advisable to use a moderately high MeshCutoff when invoking this option (300-500 Ry). The size of the "basin of attraction" around each atom in the Bader analysis should be monitored to check that the model core charge is contained in it.

The radii for the model core charges can be specified in the input fdf file. For example:

```
bader-core-radius-standard  1.3 Bohr
bader-core-radius-hydrogen  0.4 Bohr
```

The suggested way to run the Bader analysis with the Univ. of Texas code is to use both the RHO and BADER files (both in "cube" format), with the BADER file providing the "reference" and the RHO file the actual significant valence charge data which is important in bonding. (See the notes for pseudopotential codes in the above web page.) For example, for the h2o-pop example:

```
bader h2o-pop.RHO.cube -ref h2o-pop.BADER.cube
```

If netCDF support is compiled in, the file `BaderCharge.grid.nc` is produced.

AnalyzeChargeDensityOnly **false** *(logical)*

If **true**, the program optionally generates charge density files and computes partial atomic charges (Hirshfeld, Voronoi, Bader) from the information in the input density matrix, and stops. This is useful to analyze the properties of the charge density without a diagonalization step, and with a user-selectable mesh cutoff. Note that the **DM.UseSaveDM** option should be active. Note also that if an initial density matrix (DM file) is used, it is not normalized. All the relevant fdf options for charge-density file production and partial charge calculation can be used with this option.

SaveInitialChargeDensity **false** *(logical)*

If **true**, the program generates a `SystemLabel.RHOINIT` file (and a `RhoInit.grid.nc` file if netCDF support is compiled in) containing the charge density used to start the first self-consistency step, and it stops. Note that if an initial density matrix (DM file) is used, it is not normalized. This is useful to generate the charge density associated to "partial" DMs, as created by progras such as `dm_creator` and `dm_filter`.

(This option is to be deprecated in favor of **AnalyzeChargeDensityOnly**).

6.23 Auxiliary Force field

It is possible to supplement the DFT interactions with a limited set of force-field options, typically useful to simulate dispersion interactions. It is not yet possible to turn off DFT and base the dynamics only on the force field. The `GULP` program should be used for that.

%block MM.Potentials ⟨**None**⟩ *(block)*

This block allows the input of molecular mechanics potentials between species. The following potentials are currently implemented:

- C6, C8, C10 powers of the Tang-Toennes damped dispersion potential.
- A harmonic interaction.
- A dispersion potential of the Grimme type (similar to the C6 type but with a different damping function). (See S. Grimme, J. Comput. Chem. Vol 27, 1787-1799 (2006)). See also **MM.Grimme.D** and **MM.Grimme.S6** below.

The format of the input is the two species numbers that are to interact, the potential name (C6, C8, C10, harm, or Grimme), followed by the potential parameters. For the damped dispersion potentials the first number is the coefficient and the second is the exponent of the damping term (i.e., a reciprocal length). A value of zero for the latter term implies no damping. For the harmonic potential the force constant is given first, followed by r0. For the Grimme potential C6 is given first, followed by the (corrected) sum of the van der Waals radii for the interacting species (a real length). Positive values of the C6, C8, and C10 coefficients imply attractive potentials.

```
%block MM.Potentials
  1 1 C6 32.0 2.0
  1 2 harm 3.0 1.4
```

```
  2 3 Grimme 6.0 3.2
%endblock MM.Potentials
```

To automatically create input for Grimme's method, please see the utility: `Util/Grimme` which can read an fdf file and create the correct input for Grimme's method.

MM.Cutoff 30 Bohr *(length)*

Specifies the distance out to which molecular mechanics potential will act before being treated as going to zero.

MM.UnitsEnergy eV *(unit)*

Specifies the units to be used for energy in the molecular mechanics potentials.

MM.UnitsDistance Ang *(unit)*

Specifies the units to be used for distance in the molecular mechanics potentials.

MM.Grimme.D 20.0 *(real)*

Specifies the scale factor d for the scaling function in the Grimme dispersion potential (see above).

MM.Grimme.S6 1.66 *(real)*

Specifies the overall fitting factor s_6 for the Grimme dispersion potential (see above). This number depends on the quality of the basis set, the exchange-correlation functional, and the fitting set.

6.24 Parallel options

BlockSize ⟨**automatic**⟩ *(integer)*

The orbitals are distributed over the processors when running in parallel using a 1-D block-cyclic algorithm. **BlockSize** is the number of consecutive orbitals which are located on a given processor before moving to the next one. Large values of this parameter lead to poor load balancing, while small values can lead to inefficient execution. The performance of the parallel code can be optimised by varying this parameter until a suitable value is found.

ProcessorY ⟨**automatic**⟩ *(integer)*

The mesh points are divided in the Y and Z directions (more precisely, along the second and third lattice vectors) over the processors in a 2-D grid. **ProcessorY** specifies the dimension of the processor grid in the Y-direction and must be a factor of the total number of processors. Ideally the processors should be divided so that the number of mesh points per processor along each axis is as similar as possible.

Defaults to a multiple of number of processors.

Diag.Memory 1 *(real)*

Whether the parallel diagonalisation of a matrix is successful or not can depend on how much workspace is available to the routine when there are clusters of eigenvalues. **Diag.Memory** allows the user to increase the memory available, when necessary, to achieve successful diagonalisation and is a scale factor relative to the minimum amount of memory that SCALAPACK might need.

Diag.ParallelOverK **false** *(logical)*

For the diagonalisation there is a choice in strategy about whether to parallelise over the K points or over the orbitals. K point diagonalisation is close to perfectly parallel but is only useful where the number of K points is much larger than the number of processors and therefore orbital parallelisation is generally preferred. The exception is for metals where the unit cell is small, but the number of K points to be sampled is very large. In this last case it is recommend that this option be used.

NOTE: This scheme is not used for the diagonalizations involved in the generation of the band-structure (as specified with **BandLines** or **BandPoints**) or in the generation of wave-function information (as specified with **WaveFuncKPoints**). In these cases the program falls back to using parallelization over orbitals.

Use: Controls whether the diagonalisation is parallelised with respect to orbitals or K points – not allowed for non-co-linear spin case.

6.24.1 Parallel decompositions for O(N)

Apart from the default block-cyclic decomposition of the orbital data, O(N) calculations can use other schemes which should be more efficient: spatial decomposition (based on atom proximity), and domain decomposition (based on the most efficient abstract partition of the interaction graph of the Hamiltonian).

UseDomainDecomposition **false** *(logical)*

This option instructs the program to employ a graph-partitioning algorithm (using the `METIS` library (See `www.cs.umn.edu/~metis`) to find an efficient distribution of the orbital data over processors. To use this option (meaningful only in parallel) the program has to be compiled with the preprocessor option `SIESTA__METIS` (or the deprecated `ON_DOMAIN_DECOMP`) and the `METIS` library has to be linked in.

UseSpatialDecomposition **false** *(logical)*

When performing a parallel order N calculation, this option instructs the program to execute a spatial decomposition algorithm in which the system is divided into cells, which are then assigned, together with the orbitals centered in them, to the different processors. The size of the cells is, by default, equal to the maximum distance at which there is a non-zero matrix element in the Hamiltonian between two orbitals, or the radius of the Localized Wannier function - which ever is the larger. If this is the case, then an orbital will only interact with other orbitals in the same or neighbouring cells. However, by decreasing the cell size and searching over more cells it is possible to achieve better load balance in some cases. This is controlled by the variable **RcSpatial**.

NOTE: the distribution algorithm is quite fragile and a careful tuning of **RcSpatial** might be needed. This option is therefore not enabled by default.

RcSpatial ⟨**maximum orbital range**⟩ *(length)*

Controls the cell size during the spatial decomposition.

6.25 Efficiency options

DirectPhi **false** *(logical)*

The calculation of the matrix elements on the mesh requires the value of the orbitals on the mesh points. This array represents one of the largest uses of memory within the code. If set to true this option allows the code to generate the orbital values when needed rather than storing the values. This obviously costs more computer time but will make it possible to run larger jobs where memory is the limiting factor.

This controls whether the values of the orbitals at the mesh points are stored or calculated on the fly.

6.26 Memory, CPU-time, and Wall time accounting options

AllocReportLevel 0 *(integer)*

Sets the level of the allocation report, printed in file `SystemLabel.alloc`. However, not all the allocated arrays are included in the report (this will be corrected in future versions). The allowed values are:

- level 0 : no report at all (the default)
- level 1 : only total memory peak and where it occurred
- level 2 : detailed report printed only at normal program termination
- level 3 : detailed report printed at every new memory peak
- level 4 : print every individual (re)allocation or deallocation

NOTE: In MPI runs, only node-0 peak reports are produced.

AllocReportThreshold 0. *(real)*

Sets the minimum size (in bytes) of the arrays whose memory use is individually printed in the detailed allocation reports (levels 2 and 3). It does not affect the reported memory sums and peaks, which always include all arrays.

TimerReportThreshold 0. *(real)*

Sets the minimum fraction, of total CPU time, of the subroutines or code sections whose CPU time is individually printed in the detailed timer reports. To obtain the accounting of MPI communication times in parallel executions, you must compile with option `-DMPI_TIMING`. In serial execution, the CPU times are printed at the end of the output file. In parallel execution, they are reported in a separated file named `SystemLabel.times`.

UseTreeTimer **false** *(logical)*

Enable an experimental timer which is based on wall time on the master node and is aware of the tree-structure of the timed sections.

NOTE: , if used with the PEXSI solver (see Sec. 6.13) this defaults to **true**.

UseParallelTimer **true** *(logical)*

Determine whether timings are performed in parallel. This may introduce slight overhead.

NOTE: , if used with the PEXSI solver (see Sec. 6.13) this defaults to **false**.

MaxWalltime **Infinity** *(real time)*

Set an internal limit to the wall time allotted to the program's execution. Typically this is related to the external limit imposed by queuing systems. The code checks its wall time periodically and will abort if nearing the limit, with some slack left for clean-up operations (proper closing of files, emergency output...), as determined by **MaxWalltime.Slack**. See Sec. 18 for available units of time (**s**, **mins**, **hours**, **days**).

MaxWalltime.Slack **5 s** *(real time)*

The code checks its wall time T_{wall} periodically and will abort if $T_{\mathrm{wall}} > T_{\mathrm{max}} - T_{\mathrm{slack}}$, so that some slack is left for any clean-up operations.

6.27 The catch-all option UseSaveData

This is a dangerous feature, and is deprecated, but retained for historical compatibility. Use the individual options instead.

UseSaveData **false** *(logical)*

Instructs to use as much information as possible stored from previous runs in files `SystemLabel.XV`, `SystemLabel.DM` and `SystemLabel.LWF`,

NOTE: if the files are not existing it will read the information from the fdf file.

6.28 Output of information for Denchar

The program `denchar` in `Util/Denchar` can generate charge-density and wavefunction information in real space.

Write.Denchar **false** *(logical)*

Instructs to write information needed by the utility program DENCHAR (by J. Junquera and P. Ordejón) to generate valence charge densities and/or wavefunctions in real space (see `Util/Denchar`). The information is written in files `SystemLabel.PLD` and `SystemLabel.DIM`.

To run DENCHAR you will need, apart from the `.PLD` and `.DIM` files, the Density-Matrix (DM) file and/or a wavefunction (`.WFSX`) file, and the .ion files containing the information about the basis orbitals.

6.29 NetCDF (CDF4) output file

NOTE: this requires SIESTA compiled with CDF4 support.

To unify and construct a simple output file for an entire SIESTA calculation a generic NetCDF file will be created if SIESTA is compiled with `ncdf` support, see Sec. 2.4 and the `ncdf` section.

Generally all output to NetCDF flags, **SaveElectrostaticPotential**, etc. apply to this file as well.

One may control the output file with compressibility and parallel I/O, if needed.

CDF.Save **false** *(logical)*

Create the `SystemLabel.nc` file which is a NetCDF file.

This file will be created with a large set of *groups* which make separating the quantities easily. Also it will inherently denote the units for the stored quantities.

CDF.Compress 0 *(integer)*

Integer between 0 and 9. The former represents *no* compressing and the latter is the highest compressing.

The higher the number the more computation time is spent on compressing the data. A good compromise between speed and compression is 3.

NOTE: if one requests parallel I/O (**CDF.MPI**) this will automatically be set to 0. One cannot perform parallel IO and compress the data simultaneously.

NOTE: instead of using SIESTA for compression you may compress after execution by:

```
nccopy -d 3 -s noncompressed.nc compressed.nc
```

CDF.MPI **false** *(logical)*

Write `SystemLabel.nc` in parallel using MPI for increased performance. This has almost no memory overhead but may for very large number of processors saturate the file-system.

NOTE: this is an experimental flag.

CDF.Grid.Precision **single|double** *(string)*

At which precision should the real-space grid quantities be stored, such as the density, electrostatic potential etc.

7 STRUCTURAL RELAXATION, PHONONS, AND MOLECULAR DYNAMICS

This functionality is not SIESTA-specific, but is implemented to provide a more complete simulation package. The program has an outer geometry loop: it computes the electronic structure (and thus the forces and stresses) for a given geometry, updates the atomic positions (and maybe the cell vectors) accordingly and moves on to the next cycle. If there are molecular dynamics options missing you are highly recommend to look into **MD.TypeOfRun.Lua** or **MD.TypeOfRun.Master**.

Several options for MD and structural optimizations are implemented, selected by

MD.TypeOfRun **CG** *(string)*

CG Coordinate
optimization by conjugate gradients). Optionally (see variable **MD.VariableCell** below), the optimization can include the cell vectors.

Broyden Coordinate optimization by a modified Broyden scheme). Optionally, (see variable **MD.VariableCell** below), the optimization can include the cell vectors.

FIRE Coordinate optimization by Fast Inertial Relaxation Engine (FIRE) (E. Bitzek et al, PRL 97, 170201, (2006)). Optionally, (see variable **MD.VariableCell** below), the optimization can include the cell vectors.

Verlet Standard Verlet algorithm MD

Nose MD with temperature controlled by means of a Nosé thermostat

ParrinelloRahman MD with pressure controlled by the Parrinello-Rahman method

NoseParrinelloRahman MD with temperature controlled by means of a Nosé thermostat and pressure controlled by the Parrinello-Rahman method

Anneal MD with annealing to a desired temperature and/or pressure (see variable **MD.AnnealOption** below)

FC Compute force constants matrix for phonon calculations.

Master|Forces Receive coordinates from, and return forces to, an external driver program, using MPI, Unix pipes, or Inet sockets for communication. The routines in module `fsiesta` allow the user's program to perform this communication transparently, as if SIESTA were a conventional force-field subroutine. See `Util/SiestaSubroutine/README` for details. WARNING: if this option is specified without a driver program sending data, siesta may hang without any notice.

See directory Util/Scripting for other driving options.

Lua Fully control the MD cycle and convergence path using an external Lua script.

With an external Lua script one may control nearly everything from a script. One can query *any* internal data-structures in SIESTA and, similarly, return *any* data thus overwriting the internals. A list of ideas which may be implemented in such a Lua script are:

- New geometry relaxation algorithms
- NEB calculations
- New MD routines
- Convergence tests of **MeshCutoff** and **kgrid.MonkhorstPack**, or other parameters (currently basis set optimizations cannot be performed in the Lua script).

Sec. 11 for additional details (and a description of `flos` which implements some of the above mentioned items).

Using this option requires the compilation of SIESTA with the `flook` library.If SIESTA is not compiled as prescribed in Sec. 2.4 this option will make SIESTA die.

NOTE: if **Compat.Pre-v4-Dynamics** is **true** this will default to **Verlet**.

Note that some options specified in later variables (like quenching) modify the behavior of these MD options.

Appart from being able to act as a force subroutine for a driver program that uses module fsiesta, SIESTA is also prepared to communicate with the i-PI code (see `http://epfl-cosmo.github.io/gle4md/index.html?page=ipi`). To do this, SIESTA must be started after i-PI (it acts as a client of i-PI, communicating with it through Inet or Unix sockets), and the following lines must be present in the .fdf data file:

```
MD.TypeOfRun      Master     # equivalent to 'Forces'
Master.code       i-pi       # ( fsiesta | i-pi )
Master.interface  socket     # ( pipes | socket | mpi )
Master.address    localhost  # or driver's IP, e.g. 150.242.7.140
Master.port       10001      # 10000+siesta_process_order
Master.socketType inet       # ( inet | unix )
```

7.1 Compatibility with pre-v4 versions

Starting in the summer of 2015, some changes were made to the behavior of the program regarding default dynamics options and choice of coordinates to work with during post-processing of the electronic structure. The changes are:

- The default dynamics option is "CG" instead of "Verlet".
- The coordinates, if moved by the dynamics routines, are reset to their values at the previous step for the analysis of the electronic structure (band structure calculations, DOS, LDOS, etc).
- Some output files reflect the values of the "un-moved" coordinates.
- The default convergence criteria is now *both* density and Hamiltonian convergence, see **SCF.DM.Converge** and **SCF.H.Converge**.

To recover the previous behavior, the user can turn on the compatibility switch **Compat.Pre-v4-Dynamics**, which is off by default.

Note that complete compatibility cannot be perfectly guaranteed.

7.2 Structural relaxation

In this mode of operation, the program moves the atoms (and optionally the cell vectors) trying to minimize the forces (and stresses) on them.

These are the options common to all relaxation methods. If the Zmatrix input option is in effect (see Sec. 6.4.2) the Zmatrix-specific options take precedence. The 'MD' prefix is misleading but kept for historical reasons.

MD.VariableCell **false** *(logical)*

If **true**, the lattice is relaxed together with the atomic coordinates. It allows to target hydrostatic pressures or arbitrary stress tensors. See **MD.MaxStressTol**, **MD.TargetPressure**, **MD.TargetStress**, **MD.ConstantVolume**, and **MD.PreconditionVariableCell**.

NOTE: only compatible with **MD.TypeOfRun CG**, **Broyden** or **fire**.

MD.ConstantVolume **false** *(logical)*

If **true**, the cell volume is kept constant in a variable-cell relaxation: only the cell shape and the atomic coordinates are allowed to change. Note that it does not make much sense to specify a target stress or pressure in this case, except for anisotropic (traceless) stresses. See **MD.VariableCell**, **MD.TargetStress**.

NOTE: only compatible with **MD.TypeOfRun CG**, **Broyden** or **fire**.

MD.RelaxCellOnly **false** *(logical)*

If **true**, only the cell parameters are relaxed (by the Broyden or FIRE method, not CG). The atomic coordinates are re-scaled to the new cell, keeping the fractional coordinates constant. For **Zmatrix** calculations, the fractional position of the first atom in each molecule is kept fixed, and no attempt is made to rescale the bond distances or angles.

NOTE: only compatible with **MD.TypeOfRun Broyden** or **fire**.

MD.MaxForceTol 0.04 eV/Ang *(force)*

Force tolerance in coordinate optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** (see **MD.MaxStressTol** for variable cell).

MD.MaxStressTol 1 GPa *(pressure)*

Stress tolerance in variable-cell CG optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** and the maximum stress component is smaller than **MD.MaxStressTol**.

Special consideration is needed if used with Sankey-type basis sets, since the combination of orbital kinks at the cutoff radii and the finite-grid integration originate discontinuities in the stress components, whose magnitude depends on the cutoff radii (or energy shift) and the mesh cutoff. The tolerance has to be larger than the discontinuities to avoid endless optimizations if the target stress happens to be in a discontinuity.

MD.NumCGsteps 0 *(integer)*

Maximum number of conjugate gradient (or Broyden) minimization moves (the minimization will stop if tolerance is reached before; see **MD.MaxForceTol** below).

MD.MaxCGDispl 0.2 Bohr *(length)*

Maximum atomic displacements in an optimization move.

In the Broyden optimization method, it is also possible to limit indirectly the *initial* atomic displacements using **MD.Broyden.Initial.Inverse.Jacobian**. For the **FIRE** method, the same result can be obtained by choosing a small time step.

Note that there are Zmatrix-specific options that override this option.

MD.PreconditionVariableCell 5 Ang *(length)*

A length to multiply to the strain components in a variable-cell optimization. The strain components enter the minimization on the same footing as the coordinates. For good efficiency, this length should make the scale of energy variation with strain similar to the one due to atomic displacements. It is also used for the application of the **MD.MaxCGDispl** value to the strain components.

ZM.ForceTolLength 0.00155574 Ry/Bohr *(force)*

Parameter that controls the convergence with respect to forces on Z-matrix lengths

ZM.ForceTolAngle 0.00356549 Ry/rad *(torque)*

Parameter that controls the convergence with respect to forces on Z-matrix angles

ZM.MaxDisplLength 0.2 Bohr *(length)*

Parameter that controls the maximum change in a Z-matrix length during an optimisation step.

ZM.MaxDisplAngle 0.003 rad *(angle)*

Parameter that controls the maximum change in a Z-matrix angle during an optimisation step.

7.2.1 Conjugate-gradients optimization

This was historically the default geometry-optimization method, and all the above options were introduced specifically for it, hence their names. The following pertains only to this method:

MD.UseSaveCG **false** *(logical)*

Instructs to read the conjugate-gradient hystory information stored in file `SystemLabel.CG` by a previous run.

NOTE: to get actual continuation of iterrupted CG runs, use together with **MD.UseSaveXV true** with the `.XV` file generated in the same run as the CG file. If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

NOTE: no such feature exists yet for a Broyden-based relaxation.

7.2.2 Broyden optimization

It uses the modified Broyden algorithm to build up the Jacobian matrix. (See D.D. Johnson, PRB 38, 12807 (1988)). (Note: This is not BFGS.)

MD.Broyden.History.Steps 5 *(integer)*

Number of relaxation steps during which the modified Broyden algorithm builds up the Jacobian matrix.

MD.Broyden.Cycle.On.Maxit **true** *(logical)*

Upon reaching the maximum number of history data sets which are kept for Jacobian estimation, throw away the oldest and shift the rest to make room for a new data set. The alternative is to re-start the Broyden minimization algorithm from a first step of a diagonal inverse Jacobian (which might be useful when the minimization is stuck).

MD.Broyden.Initial.Inverse.Jacobian 1 *(real)*

Initial inverse Jacobian for the optimization procedure. (The units are those implied by the internal Siesta usage. The default value seems to work well for most systems.

7.2.3 FIRE relaxation

Implementation of the Fast Inertial Relaxation Engine (FIRE) method (E. Bitzek et al, PRL 97, 170201, (2006) in a manner compatible with the CG and Broyden modes of relaxation. (An older implementation activated by the **MD.FireQuench** variable is still available).

MD.FIRE.TimeStep ⟨**MD.LengthTimeStep**⟩ *(time)*

The (fictitious) time-step for FIRE relaxation. This is the main user-variable when the option **FIRE** for **MD.TypeOfRun** is active.

NOTE: the default value is encouraged to be changed as the link to **MD.LengthTimeStep** is misleading.

There are other low-level options tunable by the user (see the routines `fire_optim` and `cell_fire_optim` for more details.

7.3 Target stress options

Useful for structural optimizations and constant-pressure molecular dynamics.

MD.TargetPressure 0 GPa *(pressure)*

Target pressure for Parrinello-Rahman method, variable cell optimizations, and annealing options.

NOTE: this is only compatible with **MD.TypeOfRun ParrinelloRahman**, **NoseParrinelloRahman**, **CG**, **Broyden** or **FIRE** (variable cell), or **Anneal** (if **MD.AnnealOption Pressure** or **TemperatureandPressure**).

%block MD.TargetStress $-1 - 1 - 1000$ *(block)*

External or target stress tensor for variable cell optimizations. Stress components are given in a line, in the order `xx, yy, zz, xy, xz, yz`. In units of **MD.TargetPressure**, but with the opposite sign. For example, a uniaxial compressive stress of 2 GPa along the 100 direction would be given by

```
MD.TargetPressure  2. GPa
%block MD.TargetStress
    -1.0  0.0  0.0  0.0  0.0  0.0
%endblock MD.TargetStress
```

Only used if **MD.TypeOfRun** is **CG**, **Broyden** or **FIRE** and **MD.VariableCell** is **true**.

MD.RemoveIntramolecularPressure **false** *(logical)*

If **true**, the contribution to the stress coming from the internal degrees of freedom of the molecules will be subtracted from the stress tensor used in variable-cell optimization or variable-cell molecular-dynamics. This is done in an approximate manner, using the virial form of the stress, and assumming that the "mean force" over the coordinates of the molecule represents the "inter-molecular" stress. The correction term was already computed in earlier versions of SIESTA and used to report the "molecule pressure". The correction is now computed molecule-by-molecule if the Zmatrix format is used.

If the intra-molecular stress is removed, the corrected static and total stresses are printed in addition to the uncorrected items. The corrected Voigt form is also printed.

7.4 Molecular dynamics

In this mode of operation, the program moves the atoms (and optionally the cell vectors) in response to the forces (and stresses), using the classical equations of motion.

Note that the **Zmatrix** input option (see Sec. 6.4.2) is not compatible with molecular dynamics. The initial geometry can be specified using the Zmatrix format, but the Zmatrix generalized coordinates will not be updated.

MD.InitialTimeStep 1 *(integer)*

Initial time step of the MD simulation. In the current version of SIESTA it must be 1.

Used only if **MD.TypeOfRun** is not **CG** or **Broyden**.

MD.FinalTimeStep 1 *(integer)*

Final time step of the MD simulation.

MD.LengthTimeStep 1 fs *(time)*

Length of the time step of the MD simulation.

MD.InitialTemperature 0 K *(temperature/energy)*

Initial temperature for the MD run. The atoms are assigned random velocities drawn from the Maxwell-Bolzmann distribution with the corresponding temperature. The constraint of zero center of mass velocity is imposed.

NOTE: only used if **MD.TypeOfRun Verlet**, **Nose**, **ParrinelloRahman**, **NoseParrinelloRahman** or **Anneal**.

MD.TargetTemperature 0 K *(temperature/energy)*

Target temperature for Nose thermostat and annealing options.

NOTE: only used if **MD.TypeOfRun Nose**, **NoseParrinelloRahman** or **Anneal** if **MD.AnnealOption** is **Temperature** or **TemperatureandPressure**.

MD.NoseMass 100 Ry fs^2 *(moment of inertia)*

Generalized mass of Nose variable. This determines the time scale of the Nose variable dynamics, and the coupling of the thermal bath to the physical system.

Only used for Nose MD runs.

MD.ParrinelloRahmanMass 100 Ry fs^2 *(moment of inertia)*

Generalized mass of Parrinello-Rahman variable. This determines the time scale of the Parrinello-Rahman variable dynamics, and its coupling to the physical system.

Only used for Parrinello-Rahman MD runs.

MD.AnnealOption **TemperatureAndPressure** *(string)*

Type of annealing MD to perform. The target temperature or pressure are achieved by velocity and unit cell rescaling, in a given time determined by the variable **MD.TauRelax** below.

Temperature Reach a target temperature by velocity rescaling

Pressure Reach a target pressure by scaling of the unit cell size and shape

TemperatureandPressure Reach a target temperature and pressure by velocity rescaling and by scaling of the unit cell size and shape

Only applicable for **MD.TypeOfRun Anneal**.

MD.TauRelax 100 fs *(time)*

Relaxation time to reach target temperature and/or pressure in annealing MD. Note that this is a "relaxation time", and as such it gives a rough estimate of the time needed to achieve the given targets. As a normal simulation also exhibits oscillations, the actual time needed to reach the *averaged* targets will be significantly longer.

Only applicable for **MD.TypeOfRun Anneal**.

MD.BulkModulus 100 Ry/Bohr^3 *(pressure)*

Estimate (may be rough) of the bulk modulus of the system. This is needed to set the rate of change of cell shape to reach target pressure in annealing MD.

Only applicable for **MD.TypeOfRun Anneal**, when **MD.AnnealOption** is **Pressure** or **TemperatureAndPressure**

7.5 Output options for dynamics

Every time the atoms move, either during coordinate relaxation or molecular dynamics, their positions **predicted for next step** and **current** velocities are stored in file `SystemLabel.XV`. The shape of the unit cell and its associated 'velocity' (in Parrinello-Rahman dynamics) are also stored in this file.

WriteCoorInitial **true** *(logical)*

It determines whether the initial atomic coordinates of the simulation are dumped into the main output file. These coordinates correspond to the ones actually used in the first step (see the section on precedence issues in structural input) and are output in Cartesian coordinates in Bohr units.

It is not affected by the setting of **LongOutput**.

WriteCoorStep **false** *(logical)*

If **true**, it writes the atomic coordinates to standard output at every MD time step or relaxation step. The coordinates are always written in the `SystemLabel.XV` file, but overriden at every step. They can be also accumulated in the `.MD` or `SystemLabel.MDX` files depending on **WriteMDHistory**.

WriteForces **false** *(logical)*

If **true**, it writes the atomic forces to the output file at every MD time step or relaxation step. Note that the forces of the last step can be found in the file `SystemLabel.FA`. If constraints are used, the file `SystemLabel.FAC` is also written.

WriteMDHistory **false** *(logical)*

If **true**, SIESTA accumulates the molecular dynamics trajectory in the following files:

- `SystemLabel.MD` : atomic coordinates and velocities (and lattice vectors and their time derivatives, if the dynamics implies variable cell). The information is stored unformatted for postprocessing with utility programs to analyze the MD trajectory.
- `SystemLabel.MDE` : shorter description of the run, with energy, temperature, etc., per time step.

These files are accumulative even for different runs.

The trajectory of a molecular dynamics run (or a conjugate gradient minimization) can be accumulated in different files: SystemLabel.MD, SystemLabel.MDE, and SystemLabel.ANI. The first file keeps the whole trajectory information, meaning positions and velocities at every time step, including lattice vectors if the cell varies. NOTE that the positions (and maybe the cell vectors) stored at each time step are the **predicted** values for the next step. Care should be taken if joint position-velocity correlations need to be computed from this file. The second gives global information (energy, temperature, etc), and the third has the coordinates in a form suited for XMol animation. See the **WriteMDHistory** and **WriteMDXmol** data descriptors above for information. SIESTA always appends new information on these files, making them accumulative even for different runs.

The `iomd` subroutine can generate both an unformatted file `.MD` (default) or ASCII formatted files `.MDX` and `.MDC` containing the atomic and lattice trajectories, respectively. Edit the file to change the settings if desired.

7.6 Restarting geometry optimizations and MD runs

Every time the atoms move, either during coordinate relaxation or molecular dynamics, their **positions predicted for next step** and **current velocities** are stored in file SystemLabel.XV, where SystemLabel is the value of that FDF descriptor (or 'siesta' by default). The shape of the unit cell and its associated 'velocity' (in Parrinello-Rahman dynamics) are also stored in this file. For MD runs of type Verlet, Parrinello-Rahman, Nose, Nose-Parrinello-Rahman, or Anneal, a file named SystemLabel.VERLET_RESTART, SystemLabel.PR_RESTART, SystemLabel.NOSE_RESTART, SystemLabel.NPR_RESTART, or SystemLabel.ANNEAL_RESTART, respectively, is created to hold the values of auxiliary variables needed for a completely seamless continuation.

If the restart file is not available, a simulation can still make use of the XV information, and "restart" by basically repeating the last-computed step (the positions are shifted backwards by using a single Euler-like step with the current velocities as derivatives). While this feature does not result in seamless continuations, it allows cross-restarts (those in which a simulation of one kind (e.g., Anneal) is followed by another (e.g., Nose)), and permits to re-use dynamical information from old runs.

This restart fix is not satisfactory from a fundamental point of view, so the MD subsystem in Siesta will have to be redesigned eventually. In the meantime, users are reminded that the scripting hooks being steadily introduced (see Util/Scripting) might be used to create custom-made MD scripts.

7.7 Use of general constraints

Note: The Zmatrix format (see Sec. 6.4.2) provides an alternative constraint formulation which can be useful for system involving molecules.

%block Geometry.Constraints ⟨**None**⟩ *(block)*

Constrains certain atomic coordinates or cell parameters in a consistent method.

There are a high number of configurable parameters that may be used to control the relaxation of the coordinates.

NOTE: SIESTA prints out a small section of how the constraints are recognized.

atom|position Fix certain atomic coordinates.

This option takes a variable number of integers which each correspond to the atomic index (or input sequence) in **AtomicCoordinatesAndAtomicSpecies**.

atom is now the preferred input option while **position** still works for backwards compatibility.

One may also specify ranges of atoms according to:

atom ***A*** [***B*** [***C*** [**...**]]] A sequence of atomic indices which are constrained.

atom from ***A*** **to** ***B*** [**step** ***s***] Here atoms A up to and including B are constrained. If **step <s>** is given, the range A:B will be taken in steps of s.

```
atom from 3 to 10 step 2
```

will constrain atoms 3, 5, 7 and 9.

atom from *A* plus/minus *B* [step *s*] Here atoms A up to and including $A + B - 1$ are constrained. If **step <s>** is given, the range $A{:}A + B - 1$ will be taken in steps of s.

atom [*A*, *B* -- *C* [step *s*], *D*] Equivalent to **from . . . to** specification, however in a shorter variant. Note that the list may contain arbitrary number of ranges and/or individual indices.

```
atom [2, 3 -- 10 step 2, 6]
```

will constrain atoms 2, 3, 5, 7, 9 and 6.

```
atom [2, 3 -- 6, 8]
```

will constrain atoms 2, 3, 4, 5, 6 and 8.

atom all Constrain all atoms.

NOTE: these specifications are apt for *directional* constraints.

Z Equivalent to **atom** with all indices of the atoms that have atomic number equal to the specified number.

NOTE: this specification is apt for *directional* constraints.

species-i Equivalent to **atom** with all indices of the atoms that have species according to the **ChemicalSpeciesLabel** and **AtomicCoordinatesAndAtomicSpecies**.

NOTE: this specification is apt for *directional* constraints.

center One may retain the coordinate center of a range of atoms (say molecules or other groups of atoms).

Atomic indices may be specified according to **atom**.

NOTE: this specification is apt for *directional* constraints.

rigid|molecule Move a selection of atoms together as though they where one atom.

The forces are summed and averaged to get a net-force on the entire molecule.

Atomic indices may be specified according to **atom**.

NOTE: this specification is apt for *directional* constraints.

rigid-max|molecule-max Move a selection of atoms together as though they where one atom.

The maximum force acting on one of the atoms in the selection will be expanded to act on all atoms specified.

Atomic indices may be specified according to **atom**.

cell-angle Control whether the cell angles (α, β, γ) may be altered.

This takes either one or more of **alpha/beta/gamma** as argument.

alpha is the angle between the 2nd and 3rd cell vector.

beta is the angle between the 1st and 3rd cell vector.

gamma is the angle between the 1st and 2nd cell vector.

NOTE: currently only one angle can be constrained at a time and it forces only the spanning vectors to be relaxed.

cell-vector Control whether the cell vectors (A, B, C) may be altered.

This takes either one or more of **A/B/C** as argument.

Constraining the cell-vectors are only allowed if they only have a component along their respective Cartesian direction. I.e. **B** must only have a y-component.

stress Control which of the 6 stress components are constrained.

This takes a number of integers $1 \leq i \leq 6$ where 1 corresponds to the AA stress-component, 2 is BB, 3 is CC, 4 is BC/CB, 5 is AC/CA and 6 is AB/BA.

routine This calls the `constr` routine specified in the file: `constr.f`. Without having changed the corresponding source file, this does nothing. See details and comments in the source-file.

clear Remove constraints on selected atoms from all previously specified constraints.

This may be handy when specifying constraints via **Z** or **species-i**.

Atomic indices may be specified according to **atom**.

clear-prev Remove constraints on selected atoms from the *previous* specified constraint.

This may be handy when specifying constraints via **Z** or **species-i**.

Atomic indices may be specified according to **atom**.

NOTE: two consecutive **clear-prev** may be used in conjunction as though the atoms where specified on the same line.

It is instructive to give an example of the input options presented.

Consider a benzene molecule (C_6H_6) and we wish to relax all Hydrogen atoms. This may be accomplished in this fashion

```
%block Geometry.Constraints
  Z 6
%endblock
```

Or as in this example

```
%block AtomicCoordinatesAndAtomicSpecies
  ... ... ... 1   # C 1
  ... ... ... 2   # H 2
  ... ... ... 1   # C 3
  ... ... ... 2   # H 4
  ... ... ... 1   # C 5
  ... ... ... 2   # H 6
  ... ... ... 1   # C 7
  ... ... ... 2   # H 8
  ... ... ... 1   # C 9
  ... ... ... 2   # H 10
  ... ... ... 1   # C 11
  ... ... ... 2   # H 12
%endblock
%block Geometry.Constraints
  atom from 1 to 12 step 2
%endblock
%block Geometry.Constraints
  atom [1 -- 12 step 2]
%endblock
%block Geometry.Constraints
  atom all
  clear-prev [2 -- 12 step 2]
%endblock
```

where the 3 last blocks all create the same result.

Finally, the *directional* constraint is an important and often useful feature. When relaxing complex structures it may be advantageous to first relax along a given direction (where you expect the stress to be largest) and subsequently let it fully relax. Another example would be to relax the binding distance between a molecule and a surface, before relaxing the entire system by forcing the molecule and adsorption site to relax together. To use directional constraint one may provide an additional 3 *reals* after the **atom**/**rigid**. For instance in the previous example (benzene) one may first relax all Hydrogen atoms along the y and z Cartesian vector by constraining the x Cartesian vector

```
%block Geometry.Constraints
  Z 6 1. 0. 0.
%endblock
```

Note that you *must* append a "." to denote it a real. The vector specified need not be normalized. Also, if you want it to be constrained along the x-y vector you may do

```
%block Geometry.Constraints
  Z 6 1. 1. 0.
%endblock
```

7.8 Phonon calculations

If **MD.TypeOfRun** is **FC**, SIESTA sets up a special outer geometry loop that displaces individual atoms along the coordinate directions to build the force-constant matrix.

The output (see below) can be analyzed to extract phonon frequencies and vectors with the VIBRA package in the `Util/Vibra` directory. For computing the Born effective charges together with the force constants, see **BornCharge**.

MD.FCDispl 0.04 Bohr *(length)*

Displacement to use for the computation of the force constant matrix for phonon calculations.

MD.FCFirst 1 *(integer)*

Index of first atom to displace for the computation of the force constant matrix for phonon calculations.

MD.FCLast ⟨**MD.FCFirst**⟩ *(integer)*

Index of last atom to displace for the computation of the force constant matrix for phonon calculations.

The force-constants matrix is written in file `SystemLabel.FC`. The format is the following: for the displacement of each atom in each direction, the forces on each of the other atoms is writen (divided by the value of the displacement), in units of eV/Å^2. Each line has the forces in the x, y and z direction for one of the atoms.

If constraints are used, the file `SystemLabel.FCC` is also written.

8 LINRES

NOTE: NEW FEATURE UNDER CONSTRUCTION ...

SIESTA now includes the linear response theory, an implementation of the Density Functional Perturbation Theory (DFPT). The present implementation of DFPT was coded with the aim of calculating the vibrational properties of clusters and solids via the Force Constant matrix (FC), i.e. the second derivatives of the energy with respect to atomic displacements.

This implementation is based on the old LINRES implementation made by J. M. Pruneda, J. Junquera and P. Ordejón for the SIESTA-0.12 version (J. M. Pruneda, *Phd. Thesis*, Universidad de Oviedo (2002)).

For the calculation of the FC matrix, a standar SIESTA run is performed in order to calculate the unperturbed density matrix. Once the system reaches the self-consistent solution, the LINRES method is called. The density matrix is calculated using DFPT for each perturbed atom (external loop over the perturbed atoms) and the contributions to the second energy derivative are added. It is important to note that the calculated perturbed density matrix for each perturbed atom is independent of the rest. Once the atom loop ends, the FC matrix is written to a file SystemLabel.FC as usual. The output can be analyzed to extract phonon frequencies and vectors with the VIBRA package in the `Util/Vibra`.

During a LINRES calculation, a temporary file SystemLabel.LRDYNMAT is created. This file contains the not-fully-complete FC matrix and the last moved atom which has included its contribution to the FC matrix in order to restart the calculation if it stops unexpectedly.

MD.TypeOfRun **LR** *(string)*

Input to perform a LINRES calculation after the single point.

MD.FCFirst 1 *(integer)*

Index of first atom to displace for the computation of the force constant matrix for phonon calculations.

MD.FCLast ⟨**MD.FCFirst**⟩ *(integer)*

Index of last atom to displace for the computation of the force constant matrix for phonon calculations.

LR.DMTolerance 10^{-3} *(real)*

Tolerance of the perturbed Density Matrix to achieve the self-consistency (as in the standard way with the Density matrix).

LR.EigTolerance 10^{-3} eV *(real)*

Energy difference between energy levels to consider that there is a degeneracy.

LR.readDynmat **false** *(logical)*

Restart calculation from SystemLabel.LRDYNMAT file.

Note: as it is implemented in previous SIESTA versions, the self-consistency is achieved using the Pulay mixing scheme. Check the Pulay options in the appropriate section.

9 LDA+U

Important note: Current implementation is based on the simplified rotationally invariant LDA+U formulation of Dudarev and collaborators [see, Dudarev *et al.*, Phys. Rev. B **57**, 1505 (1998)]. Although the input allows to define independent values of the U and J parameters for each atomic shell, in the actual calculation the two parameters are combined to produce an effective Coulomb repulsion $U_{\text{eff}} = U - J$. U_{eff} is the parameter actually used in the calculations for the time being.

For large or intermediate values of U_{eff} the convergence is sometimes difficult. A step-by-step increase of the value of U_{eff} can be advisable in such cases.

Currently, the LDA+U implementation does not support non-colinear, nor spin-orbit coupling.

LDAU.ProjectorGenerationMethod **2** *(integer)*

Generation method of the LDA+U projectors. The LDA+U projectors are the localized functions used to calculate the local populations used in a Hubbard-like term that modifies the LDA Hamiltonian and energy. It is important to recall that LDA+U projectors should be quite localized functions. Otherwise the calculated populations loose their atomic character and physical meaning. Even more importantly, the interaction range can increase so much that jeopardizes the efficiency of the calculation.

Two methods are currently implemented:

1 Projectors are slightly-excited numerical atomic orbitals similar to those used as an automatic basis set by SIESTA. The radii of these orbitals are controlled using the parameter **LDAU.EnergyShift** and/or the data included in the block **LDAU.Proj** (quite similar to the data block **PAO.Basis** used to specify the basis set, see below).

2 Projectors are exact solutions of the pseudoatomic problem (and, in principle, are not strictly localized) which are cut using a Fermi function $1/\{1 + \exp[(r - r_c)\omega]\}$. The values of r_c and ω are controlled using the parameter **LDAU.CutoffNorm** and/or the data included in the block **LDAU.Proj**.

LDAU.EnergyShift 0.05 Ry *(energy)*

Energy increase used to define the localization radius of the LDA+U projectors (similar to the parameter **PAO.EnergyShift**).

NOTE: only used when **LDAU.ProjectorGenerationMethod** is **1**.

LDAU.CutoffNorm 0.9 *(real)*

Parameter used to define the value of r_c used in the Fermi distribution to cut the LDA+U projectors generated according to generation method 2 (see above). **LDAU.CutoffNorm** is the norm of the original pseudoatomic orbital contained inside a sphere of radius equal to r_c.

NOTE: only used when **LDAU.ProjectorGenerationMethod** is **2**.

%block LDAU.Proj ⟨**None**⟩ *(block)*

Data block used to specify the LDA+U projectors.

- If **LDAU.ProjectorGenerationMethod** is **1**, the syntax is as follows:

```
%block LDAU.Proj      # Define LDA+U projectors
 Fe    2              # Label, l_shells
  n=3 2  E 50.0 2.5   # n (opt if not using semicore levels),l,Softconf(opt)
```

```
      5.00  0.35      # U(eV), J(eV) for this shell
      2.30            # rc (Bohr)
      0.95            # scaleFactor (opt)
      0               #    l
      1.00  0.05      # U(eV), J(eV) for this shell
      0.00            # rc(Bohr) (if 0, automatic r_c from LDAU.EnergyShift)
%endblock LDAU.Proj
```

- If **LDAU.ProjectorGenerationMethod** is **2**, the syntax is as follows:

```
%block LDAU.Proj      # Define LDAU projectors
 Fe    2              # Label, l_shells
  n=3 2  E 50.0 2.5   # n (opt if not using semicore levels),l,Softconf(opt)
      5.00  0.35      # U(eV), J(eV) for this shell
      2.30  0.15      # rc (Bohr), \omega(Bohr) (Fermi cutoff function)
      0.95            # scaleFactor (opt)
      0               #    l
      1.00  0.05      # U(eV), J(eV) for this shell
      0.00  0.00      # rc(Bohr), \omega(Bohr) (if 0 r_c from LDAU.CutoffNorm
%endblock LDAU.Proj   #                          and \omega from default value)
```

Certain of the quantites have default values:

U	**0.0 eV**
J	**0.0 eV**
ω	**0.05 Bohr**
Scale factor	**1.0**

r_c depends on **LDAU.EnergyShift** or **LDAU.CutoffNorm** depending on the generation method.

LDAU.FirstIteration **false** *(logical)*

If **true**, local populations are calculated and Hubbard-like term is switch on in the first iteration. Useful if restarting a calculation reading a converged or an almost converged density matrix from file.

LDAU.ThresholdTol 0.01 *(real)*

Local populations only calculated and/or updated if the change in the density matrix elements (dDmax) is lower than **LDAU.ThresholdTol**.

LDAU.PopTol 0.001 *(real)*

Convergence criterium for the LDA+U local populations. In the current implementation the Hubbard-like term of the Hamiltonian is only updated (except for the last iteration) if the variations of the local populations are larger than this value.

LDAU.PotentialShift **false** *(logical)*

If set to **true**, the value given to the U parameter in the input file is interpreted as a local potential shift. Recording the change of the local populations as a function of this potential shift, we can calculate the appropriate value of U for the system under study following the methology proposed by Cococcioni and Gironcoli in Phys. Rev. B **71**, 035105 (2005).

10 RT-TDDFT

Now it is possible to perform Real-Time Time-Dependent Density Functional Theory (RT-TDDFT)-based calculations using the SIESTA method. This section includes a brief introduction to the TDDFT method and implementation, shows how to run the TDDFT-based calculations, and provides a reference guide to the additional input options.

10.1 Brief description

The basic features of the TDDFT have been implemented within the SIESTA code. Most of the details can be found in the paper Phys. Rev. B **66** 235416 (2002), by A. Tsolakidis, D. Sánchez-Portal and, Richard M. Martin. However, the practical implementation of the present version is very different from the initial version. The present implementation of the TDDFT has been programmed with the primary aim of calculating the optical response of clusters and solids, however, it has been successfully used to calculate the electronic stopping power of solids as well.

For the calculation of the optical response of the electronic systems a perturbation to the system is applied at time step 0, and the system is allowed to reach a self-consistent solution. Then, the perturbation is switched off for all subsequent time steps, and the electrons are allowed to evolve according to time-dependent Kohn-Sham equations. For the case of a cluster the perturbation is a finite (small) electric field. For the case of bulk material (not yet fully implemented) the initial perturbation is different but the main strategy is similar.

The present version of the RT-TDDFT implementation is also capable of performing a simultaneous dynamics of electrons and ions but this is limited to the cases in which forces on the ions are within ignorable limit.

The general working scheme is as following. First, the system is allowed to reach a self-consistent solution for some initial conditions (for example an initial ionic configuration or with applied external field). This is achieved using standard diagonalization [7]. The occupied Kohn-Sham orbitals (KSOs) are then selected and stored in memory. The occupied KSOs are then made to evolve in time, and the Hamiltonian is recalculated for each time step.

10.2 Input options for RT-TDDFT

A TDDFT calculation requires two runs of SIESTA. In the first run with appropriate flags it calculates the self-consistent initial state, i.e., only occupied initial KSOs stored in `SystemLabel.TDWF` file. The second run uses this file as input and evolves the occupied KSOs.

WriteInitialTDWF **false** *(logical)*

If set to .true. in a standard self-consistent SIESTAcalculation, it makes the program save the KSOs after reaching self-consistency. This constitutes the first run.

ElectronDynamics ⟨**None**⟩ *(string)*

A new Molecular Dynamics option, **MD.TypeOfRun**, has been included. The second run of SIESTA uses this option with the file `SystemLabel.TDWF` present in the working directory. In this option ions and electrons are assumed to move simultaneously. The occupied electronic

[7] Caution! we need the Kohn-Sham orbitals.

states are time-evolved instead of the usual SCF calculations in each step. Choose this option even if you intend to do only-electron dynamics. If you want to do an electron dynamics-only calculation set **MD.FinalTimeStep** equal to 1. For optical response calculations switch off the external field during the second run. The **MD.LengthTimeStep** is, as in the standard MD simulation, used to set the time step for each MD step.

TDED.Nsteps **1** *(integer)*

Number of electronic time steps between each atomic movement. The time step for each electronic step is determined by dividing the MD time step (**MD.LengthTimeStep**) over number of electronic steps (**TDED.Nsteps**). In case of an only electron dynamics simulation the **MD.LengthTimeStep** becomes total simulation time.

TDED.Nwrite **100** *(integer)*

Fixes the number of steps of ion-electron dynamics after which the instanteneous wavefunctions are saved in file `SystemLabel.TDWF` for a possible restart. The writing of wavefunctions is not parallel, hence writing too often can affect efficiency.

WriteEtotvsTime **true** *(logical)*

If .true. the total energy for every time step is stored in the file `SystemLabel.etot_vs_time`.

WriteDipolevsTime **false** *(logical)*

If .true. a file `SystemLabel.dipol_vs_time` is created that can be further processed to calculate polarizability.

WriteEigenvsTime **false** *(logical)*

If .true. the quantities $\langle\phi(t)|H|\phi(t)\rangle$ in every time step are calculated and stored in the file `SystemLabel.eigenvalues_vs_time` (not fully implemented yet!)

TDED.Saverho **false** *(logical)*

If .true. the instantaneous time-dependent density is saved to `<istep>.TDRho` after every **TDED.Nsaverho** number of steps.

TDED.Nsaverho **50** *(integer)*

Fixes the number of steps of ion-electron dynamics after which the instantaneous time-dependent density is saved. May require a lot of disk space.

11 External control of SIESTA

Since SIESTA 4.1 an additional method of controlling the convergence and MD of SIESTA is enabled through external scripting capability. The external control comes in two variants:

- Implicit control of MD through updating/changing parameters and optimizing forces. For instance one may use a **Verlet** MD method but additionally update the forces through some external force-field to amend limitations by the **Verlet** method for your particular case. In the implicit control the molecular dynamics is controlled by SIESTA.
- Explicit control of MD. In this mode the molecular dynamics *must* be controlled in the external Lua script and the convergence of the geometry should also be controlled via this script.

The implicit control is in use if **MD.TypeOfRun** is something other than **lua**, while if the option is **lua** the explicit control is in use.

For examples on the usage of the Lua scripting engine and the power you may find the library `flos`[8], see `https://github.com/siesta-project/flos`. At the time of writing the `flos` library already implements new geometry/cell relaxation schemes and new force-constants algorithms. You are highly encouraged to use the new relaxation schemes as they may provide faster convergence of the relaxation.

Lua.Script ⟨**none**⟩ *(file)*

Specify a Lua script file which may be used to control the internal variables in SIESTA. Such a script file must contain at least one function named `siesta_comm` with no arguments.

An example file could be this (note this is Lua code):

```
-- This function (siesta_comm) is REQUIRED
function siesta_comm()

   -- Define which variables we want to retrieve from SIESTA
   get_tbl = {"geom.xa", "E.total"}

   -- Signal to SIESTA which variables we want to explore
   siesta.receive(get_tbl)

   -- Now we have the required variables,
   -- convert to a simpler variable name (not nested tables)
   -- (note the returned quantities are in SIESTA units (Bohr, Ry)
   xa = siesta.geom.xa
   Etot = siesta.E.total

   -- If we know our energy is wrong by 0.001 Ry we may now
   -- change the total energy
   Etot = Etot - 0.001

   -- Return to SIESTA the total energy such that
   -- it internally has the "correct" energy.

   siesta.E.total = Etot
   ret_tbl = {"E.total"}

   siesta.send(ret_tbl)

end
```

Within this function there are certain *states* which defines different execution points in SIESTA:

Initialization This is right after SIESTA has read the options from the FDF file. Here you may query some of the FDF options (and even change them) for your particular problem.

NOTE: `siesta.state == siesta.INITIALIZE`.

Initialize-MD Right before the SCF step starts. This point is somewhat superfluous, but is

[8]This library is implemented by Nick R. Papior to further enhance the inter-operability with SIESTA and external contributions.

necessary to communicate the actual meshcutoff used[9].

NOTE: `siesta.state == siesta.INIT_MD`.

SCF Right after SIESTA has calculated the output density matrix, and just after SIESTA has performed mixing.

NOTE: `siesta.state == siesta.SCF_LOOP`.

Forces This stage is right after SIESTA has calculated the forces.

NOTE: `siesta.state == siesta.FORCES`.

Move This state will *only* be reached if **MD.TypeOfRun** is **lua**.

If one does not return updated atomic coordinates SIESTA will reuse the same geometry as just analyzed.

NOTE: `siesta.state == siesta.MOVE`.

Analysis Just before SIESTA completes and exits.

NOTE: `siesta.state == siesta.ANALYSIS`.

Beginning with implementations of Lua scripts may be cumbersome. It is recommended to start by using `flos`, see `https://github.com/siesta-project/flos` which contains several examples on how to start implementing your own scripts. Currently `flos` implements a larger variety of relaxation schemes, for instance:

```
local flos = require "flos"
LBFGS = flos.LBFGS()
function siesta_comm()
   LBFGS:SIESTA(siesta)
end
```

which is the most minimal example of using the L-BFGS algorithm for geometry relaxation. Note that `flos` reads the parameters **MD.MaxCGDispl** and **MD.MaxForceTol** through SIESTA automatically.

NOTE: The number of available variables continues to grow and to find which quantities are accessible in Lua you may add this small code in your Lua script:

```
siesta.print_allowed()
```

which prints out a list of all accessible variables (note they are not sorted).

If there are any variables you require which are not in the list, please contact the developers.

Remark that since *anything* may be changed via Lua one may easily make SIESTA crash due to inconsistencies in the internal logic. This is because SIESTA does not check what has changed, it accepts everything *as is* and continues. Hence, one should be careful what is changed.

Lua.Debug **false** *(logical)*

Debug the Lua script mode by printing out (on stdout) information everytime SIESTA communicates with Lua.

Lua.Debug.MPI **false** *(logical)*

Debug all nodes (if in a parallel run).

[9]Remember that the **MeshCutoff** defined is the minimum cutoff used.

11.1 Examples of Lua programs

Please look in the `Tests/lua_*` folders where examples of basic Lua scripts are found. Below is a description of the `*` examples.

h2o Changes the mixing weight continuously in the SCF loop. This will effectively speed up convergence time if one can attain the best mixing weight per SCF-step.

si111 Change the mixing method based on certain convergence criteria. I.e. after a certain convergence one can switch to a more aggressive mixing method.

A combination of the above two examples may greatly improve convergence, however, creating a generic method to adaptively change the mixing parameters may be very difficult to implement. If you do create such a Lua script, please share it on the mailing list.

12 TRANSIESTA

SIESTA includes the possibility of performing calculations of electronic transport properties using the TranSIESTA method. This Section describes how to compile the code to be able to use these capabilities, and a reference guide to the relevant fdf options. We describe here only the additional options available for TranSIESTA calculations, while the rest of the Siesta functionalities and variables are described in the previous sections of this User's Guide.

12.1 Brief description

The TranSIESTA method is a procedure to solve the electronic structure of an open system formed by a finite structure sandwiched between two semi-infinite metallic leads. A finite bias can be applied between both leads, to drive a finite current. The method is described in detail in [**?**]. In practical terms, calculations using TranSIESTA involve the solution of the electronic density from the DFT Hamiltonian using Greens functions techniques, instead of the usual diagonalization procedure. Therefore, TranSIESTA calculations involve a SIESTA run, in which a set of routines are invoked to solve the Greens functions and the charge density for the open system. These routines are packed in a set of modules, and we will refer to it as the 'TranSIESTA module' in what follows.

TranSIESTA was originally developed by Mads Brandbyge, José-Luis Mozos, Pablo Ordejón, Jeremy Taylor and Kurt Stokbro (see references). It consisted, mainly, in setting up an interface between SIESTA and the (tight-binding) transport codes developed by M. Brandbyge and K. Stokbro. Initially everything was written in Fortran-77. As SIESTA started to be translated to Fortran-90, so were the TranSIESTA parts of the code. This was accomplished by José-Luis Mozos, who also worked on the parallelization of TranSIESTA. Subsequently Frederico D. Novaes extended TranSIESTA to allow k-point sampling for transverse directions. Additional extensions was added by Nick R. Papior during 2012.

The current TranSIESTA module has been completely rewritten by Nick R. Papior and encompass a highly advanced inversion algorithm as well as allowing N electrode setups among just a few additional features. Furthermore, the utility TBtrans has also been fully re-coded (by Nick R. Papior) to be a generic tight-binding code capable of analyzing physics from the Greens function perspective in $N \geq 1$ setups.

12.2 Source code structure

In this implementation, the TranSIESTA routines have been grouped in a set of modules whose file names begin with `m_ts`. The inclusion of TranSIESTA has also required the modification of some of the SIESTA routines. Presently, these modifications are controlled by pre-processor compilation directives (such as in `#ifdef TRANSIESTA`). See the next section for compilation instructions.

12.3 Compilation

The standard SIESTA executable (obtained as described in Section 2) does not include the TranSIESTA modules. In order to use the TranSIESTA capabilities, you must compile the SIESTA package as indicated in this Section. In this way, the compilation is done using the appropriate preprocessor flags needed to include the TranSIESTA modules in the binary file. To generate a binary of SIESTA which includes the TranSIESTA capabilities, just type:

```
$ make transiesta
```

using the appropriate `arch.make` file for your system (note that you do not need to make any modification on your arch.make file: you can use the same one that you have used to make a standard SIESTA compilation in your system). The Makefile takes care of defining the appropriate preprocessor flag `-DTRANSIESTA` so that the TranSIESTA modules and modifications are compiled and incorporated into the binary. Upon successful compilation, the binary file `transiesta` will be generated, containing an executable version of SIESTA with TranSIESTA capabilities.

12.4 Brief explanation

- Transport calculations involve *electrode* (EL) calculations, and then the Scattering Region (SR) calculation. The *electrode* calculations are usual SIESTA calculations, but where a file `SystemLabel.TSHS` is generated. These files contain the information necessary for calculation of the self-energies. If any electrodes have identical structures (see below) the same `SystemLabel.TSHS` file can be used to describe those. In general, however, electrodes can be different and therefore two different `SystemLabel.TSHS` files must be generated. The location of these electrode files must be specified in the FDF input file of the SR calculation.

- For the SR, TranSIESTA starts with the usual SIESTA procedure, converging a Density Matrix (DM) with the usual Kohn-Sham scheme for periodic systems. It uses this solution as an initial input for the Greens function self consistent cycle. As it is known, SIESTA stores the DM in a file with extension `SystemLabel.DM`. In the case of TranSIESTA, this is done in a file named `SystemLabel.TSDE`. In a rerun of the same system (meaning the same **SystemLabel**), if the code finds a `SystemLabel.TSDE` file in the directory, it will take this DM as the initial input and this is then considered a continuation run. In this case it does not perform an initial SIESTA run. It must be clear that when starting a calculation from scratch, in the end one will find both files, `SystemLabel.DM` and `SystemLabel.TSDE`. The first one stores the SIESTA density matrix (periodic boundary conditions in all directions and no voltage), and the latter the TranSIESTA solution.

- When performing several bias calculations, it is heavily advised to copy the `SystemLabel.TSDE` for the closest, previously, calculated bias.
- The `SystemLabel.TSDE` may be read equivalently as the `SystemLabel.DM`. Thus, it may be used by fx. `denchar` to analyze the non-equilibrium charge density.
- As in the case of SIESTA calculations, what TranSIESTA does is to obtain a converged DM, but for open boundary conditions and possibly a finite bias applied between the electrodes. The corresponding Hamiltonian matrix (once self consistency is achieved) of the SR is also stored in a `SystemLabel.TSHS` file. Subsequently, transport properties are obtained in a post-processing procedure using the TBtrans code (located in the `Util/TS/TBtrans` directory). It is to be noted that the `SystemLabel.TSHS` files contain all the needed structural information (atomic positions, matrix elements, ...), and so this kind of parameters will not be changed by input (fdf) flags once they are read a `SystemLabel.TSHS` file.
- When the non-equilibrium calculation uses different electrodes one may use so-called *buffer* atoms behind the electrodes to act as additional screening region when calculating the initial guess (using SIESTA) for TranSIESTA. Essentially they may be used to achieve a better "bulk-like" environment at the electrodes.
- An important parameter is the lower bound of the energy contours. It is a good practice, to start with a SIESTA calculation for the SR and look at the eigenvalues of the system.
- TranSIESTA still assumes periodic boundary conditions in the `xy` directions. For TranSIESTA, the specified k-point sampling (of this 2-dimensional Brillouin zone) used in a SR calculation must be the same as the one that was used for the electrodes, if they are different the code will stop.
- Importantly the k-point sampling need typically be much higher in a TBtrans calculation to achieve a converged transmission function.

12.5 Electrodes

In order to calculate the electronic structure of a system under external bias, TranSIESTA attaches the system to semi-infinite electrodes which extend to their respective semi-infinite directions. Examples of electrodes would include surfaces, nanowires, nanotubes or even atomic chains. The electrode must be large enough (in the semi-infinite direction) so that orbitals within the unit cell only interact with a single nearest neighbor cell in the semi-infinite direction (the size of the unit cell can thus be derived from the range of support for the orbital basis functions). TranSIESTA will warn if this is not enforced. The electrodes are generated by a separate transiesta run on a bulk system. The results are saved in a file with extension `SystemLabel.TSHS` which contains a description of the electrode unit cell, the position of the atoms within the unit cell, as well as the Hamiltonian and overlap matrices that describe the electronic structure of the lead. One can generate a variety of electrodes and the typical use of transiesta would involve reusing the same electrode for several setups. At runtime, the transiesta coordinates are checked against the electrode coordinates and the program stops if there is a mismatch to a certain precision (10^{-4} Bohr).

Principal layer interactions It is *extremely* important that the electrodes only interact with one neighboring supercell due to the self-energy calculation. TranSIESTA will print out a block as this

```
<> principal cell is perfect!
```

if the electrode is correctly setup and it only interacts with its neighboring supercell. In case the electrode is erroneously setup, something similar to the following will be shown in the output file.

```
<> principal cell is extending out with 96 elements:
   Atom 1 connects with atom 3
   Orbital 8 connects with orbital 26
   Hamiltonian value: |H(8,6587)|@R=-2 =  0.651E-13 eV
   Overlap          :  S(8,6587)|@R=-2 =   0.00
```

It is imperative that you have a *perfect* electrode as otherwise nonphysical results will occur.

By default TranSIESTA will die if there are connections beyond the principal cell. One may control whether this is allowed or not by using **TS.Elecs.Neglect.Principal**.

12.6 TranSIESTA Options

The fdf options shown here are only to be used at the input file for the scattering region. When using TranSIESTA for electrode calculations, only the usual SIESTA options are relevant. Note that since TranSIESTA is a generic N electrode NEGF code the input options are heavily changed.

12.6.1 Quick and dirty

Since 4.1, TranSIESTA has been fully re-implemented. And so have *every* input fdf-flag. To accommodate an easy transition between previous input files and the new version format a small utility called `ts2ts`. It may be compiled in `Util/TS/ts2ts`. It is recommended that you use this tool if you are familiar with previous TranSIESTA versions.

One may input options as in the old TranSIESTA version and then run

```
ts2ts OLD.fdf > NEW.fdf
```

which translates all keys to the new, equivalent, input format. If you are familiar with the old-style flags this is highly recommendable to become comfortable with the new input format. Please note that some defaults have changed to more conservative values in the newer release.

If one does not know the old flags and wish to get a basic example of an input file, a program `Util/TS/tselecs.sh` exists that can create the basic input for N electrodes. One may call it like:

```
tselecs.sh -2 > TWO_ELECTRODE.fdf
tselecs.sh -3 > THREE_ELECTRODE.fdf
tselecs.sh -4 > FOUR_ELECTRODE.fdf
...
```

where the first call creates an input fdf for 2 electrode setups, the second for a 3 electrode setup, and so on. See the help (`-h`) for the program for additional options.

Before endeavoring on large scale calculations you are advised to run an analyzation of the system at hand, you may run your system as

```
transiesta -fdf TS.Analyze RUN.fdf > analyze.out
```

which will analyze the sparsity pattern and print out several different pivoting schemes. Please see **TS.Analyze** for additional information.

12.6.2 General options

SolutionMethod **transiesta** *(string)*

To run TranSIESTA the solution method *must* be **transiesta**.

TS.SolutionMethod **btd|mumps|full** *(string)*

Control the algorithm used for calculating the Green function. Generally the BTD method is the fastest and this option need not be changed.

TS.Voltage 0 eV *(energy)*

Define the reference applied bias. For $N = 2$ electrode calculations this refers to the actual potential drop between the electrodes, while for $N \neq 2$ this is a reference bias. In the latter case it *must* be equivalent to the maximum difference between the chemical potential of any two electrodes.

TS.Atoms.Buffer ⟨**None**⟩ *(block/list)*

Specify atoms that will be removed in the TranSIESTA SCF. They are not considered in the calculation and may be used to improve the initial guess for the Hamiltonian. Their main usage is to extend electrodes in their semi-infinite directions.

NOTE: all lines are additive for the buffer atoms and the input method is similar to that of **Geometry.Constraints** for the **atom** line(s).

```
%block TBT.Atoms.Buffer
   atom [ 1 -- 5 ]
%endblock
# Or equivalently as a list
TBT.Atoms.Buffer [1 -- 5]
```

will remove atoms [1–5] from the calculation.

TS.ElectronicTemperature ⟨**ElectronicTemperature**⟩ *(energy)*

Define the temperature used for the Fermi distributions for the chemical potentials. See **TS.ChemPot.<>.ElectronicTemperature**.

TS.SCF.Initialize **diagon|transiesta** *(string)*

Control which initial guess should be used for TranSIESTA. The general way is the **diagon** solution method, however, one can start immediately in a TranSIESTA run. If you start directly with TranSIESTA please refer to these flags: **TS.Elecs.DM.Init**, **DM.Init.Bulk** and **TS.Fermi.Initial**.

NOTE: Setting this to **transiesta** is highly experimental and convergence may be extremely poor.

TS.Fermi.Initial $\sum_i^{N_E} E_F^i/(N_E+1)$ *(energy)*

Manually set the initial Fermi level to a predefined value.

NOTE: this may also be used to change the Fermi level for calculations where you restart calculations. Using this feature is highly experimental.

TS.Weight.Method **orb-orb|[[un]correlated+][sum|tr]-atom-[atom|orb]|mean** *(string)*

Control how the NEGF weighting scheme is conducted. Generally one should only use the **orb-orb** while the others are present for more advanced usage. They refer to how the weighting coefficients of the different non-equilibrium contours are performed. In the following the weight are denoted in a two-electrode setup while they are generalized for multiple electrodes.

Define the normalised geometric mean as $\propto^{||}$ via

$$w \overset{||}{\propto} \langle .^L \rangle \equiv \frac{\langle .^L \rangle}{\langle .^L \rangle + \langle .^R \rangle}. \tag{19}$$

orb-orb Weight each orbital-density matrix element individually.

tr-atom-atom Weight according to the trace of the atomic density matrix sub-blocks

$$w_{ij}^{\mathrm{Tr}} \overset{||}{\propto} \sqrt{\sum_{\in\{i\}} (\Delta\rho_{\mu\mu}^L)^2 \sum_{\in\{j\}} (\Delta\rho_{\mu\mu}^L)^2} \tag{20}$$

tr-atom-orb Weight according to the trace of the atomic density matrix sub-block times the weight of the orbital weight

$$w_{ij,\mu\nu}^{\mathrm{Tr}} \overset{||}{\propto} \sqrt{w_{ij}^{\mathrm{Tr}} w_{ij,\mu\nu}} \tag{21}$$

sum-atom-atom Weight according to the total sum of the atomic density matrix sub-blocks

$$w_{ij,\mu\nu}^{\Sigma} \overset{||}{\propto} \sqrt{\sum_{\in\{i\}} (\Delta\rho_{\mu\nu}^L)^2 \sum_{\in\{j\}} (\Delta\rho_{\mu\nu}^L)^2} \tag{22}$$

sum-atom-orb Weight according to the total sum of the atomic density matrix sub-block times the weight of the orbital weight

$$w_{ij,\mu\nu}^{\Sigma} \overset{||}{\propto} \sqrt{w_{ij}^{\Sigma} w_{ij,\mu\nu}} \tag{23}$$

mean A standard average.

Each of the methods (except **mean**) comes in a correlated and uncorrelated variant where $\sum$ is either outside or inside the square, respectively.

TS.Weight.k.Method **correlated|uncorrelated** *(string)*

Control weighting *per* k-point or the full sum. I.e. if **uncorrelated** is used it will weight n_k times if there are n_k k-points in the Brillouin zone.

TS.Forces **true** *(logical)*

Control whether the forces are calculated. If *not* TranSIESTA will use slightly less memory and the performance slightly increased.

TS.ChargeCorrection **none|buffer|fermi** *(string)*

Any excess/deficiency of charge can be re-adjusted after each TranSIESTA cycle to reduce charge fluctuations in the cell.

none No charge corrections are introduced.

buffer Excess/missing electrons are placed in the buffer regions (buffer atoms are required to exist)

fermi Correct the filling by calculating a new Fermi-level (reference energy).
We approximate the contribution to be constant around the Fermi level and find

$$\mathrm{d}E_F = \frac{Q' - Q}{Q|_{E_F}}, \tag{24}$$

where Q' is the charge from a converged TranSIESTA calculation and $Q|_{E_F}$ is the equilibrium charge at the current Fermi level, Q is the supposed charge to reside in the calculation. Fermi correction utilizes Eq. (24) for the first correction and all subsequent corrections are based on a cubic spline interpolation to much faster converge to the "correct" Fermi level.

This method will create a file called `TS_FERMI` and only works with the BTD solver.

TS.ChargeCorrection.Factor **0.75** *(real)*

Should be between 0 and 1 to lower the charge adjustment. 0 means no charge correction. 1 means total charge conservation. This will reduce the fluctuations in the SCF and setting this to 1 may result in difficulties in converging.

TS.ChargeCorrection.Fermi.Tolerance **0.01** *(real)*

The tolerance at which the charge correction will converge. Any excess/missing charge ($|Q' - Q| > \mathrm{Tol}$) will result in a correction for the Fermi level.

TS.ChargeCorrection.Fermi.Max 1.5 eV *(energy)*

The maximally allowed value that the Fermi level will change from a charge correction using the Fermi correction method. In case the Fermi level lies in between two bands a DOS of 0 at the Fermi level will make the Fermi change equal ∞. This is not physical and the user can thus truncate the correction.

If you know the band-gab, setting this to 1/4 *(or smaller) of the band gab seems like a better value than the rather arbitrarily default one.*

TS.HS.Save **true** *(logical)*

Must be **true** for saving the Hamiltonian. In almost no cases is this useful to redefine.

TS.DE.Save **true** *(logical)*

Must be **true** for saving the density matrix for continuation runs. If **false** the `SystemLabel.TSDE` file will not be created.

TS.S.Save **false** *(logical)*

This is a flag mainly used for the Inelastica code to produce overlap matrices for Pulay corrections. This should only be used by advanced users.

TS.SIESTA.Only **false** *(logical)*

Stop TranSIESTA right after the initial diagonalization run in SIESTA. Upon exit it will also create the TSDE file which may be used for initialization runs later.

TS.Analyze **false** *(logical)*

When using the BTD solution method (**TS.SolutionMethod**) this will analyze the Hamiltonian and printout an analysis on the sparsity pattern for optimal choice of the BTD partitioning algorithm.

This yields information regarding the **TS.BTD.Pivot** flag.

NOTE: we advice users to *always* run an analyzation step prior to actual calculation and select the *best* BTD format. This analyzing step is very fast and may be performed on small work-station computers, even on systems of 10,000+ orbitals.

To run the analyzing step you may do:

```
transiesta -fdf TS.Analyze RUN.fdf > analyze.out
```

note that there is little gain on using MPI and it should complete within a few minutes, no matter the number of orbitals.

Choosing the best one may be difficult. Generally one should choose the pivoting scheme that uses the least amount of memory. However, one should also choose the method with largest block-size being as small as possible. As an example:

```
TS.BTD.Pivot atom+GPS
...
    BTD partitions (7):
     [ 2984, 2776, 192, 192, 1639, 4050, 105 ]
    Matrix elements in tri / % of full: 68246662 /  47.88707

TS.BTD.Pivot atom+GGPS
...
    BTD partitions (6):
     [ 2880, 2916, 174, 174, 2884, 2910 ]
    Matrix elements in tri / % of full: 69303556 /  48.62867
```

Although the GPS method uses the least amount of memory, the GGPS will likely perform better as the largest block in GPS is 4050 vs. 2916 for the GGPS method.

12.6.3 Algorithm specific options

These options adhere to the specific solution methods available for TranSIESTA. For instance the **TS.BTD.*** options adhere when using **TS.SolutionMethod btd**, and similarly for **MUMPS**.

TS.BTD.Pivot ⟨**first electrode**⟩ *(string)*

Decide on the partitioning for the BTD matrix. One may denote either **atom+** or **orb+** as a prefix which does the analysis on the atomic sparsity pattern or the full orbital sparsity pattern, respectively. If neither are used it will default to **atom+**.

Please see **TS.Analyze**.

<elec-name> The partitioning will be a connectivity graph starting from the electrode denoted by the name. This name *must* be found in the **TS.Elecs** block.

rev-CM Use the reverse Cuthill-McKee for pivoting the matrix elements to reduce bandwidth. One may omit **rev-** to use the standard Cuthill-McKee algorithm.

GPS Use the Gibbs-Poole-Stockmeyer algorithm for reducing the bandwidth.

GGPS Use the generalized Gibbs-Poole-Stockmeyer algorithm for reducing the bandwidth.

PCG Use the perphiral connectivity graph algorithm for reducing the bandwidth.

Examples are

```
TS.BTD.Pivot atom+GGPS
TS.BTD.Pivot GGPS
TS.BTD.Pivot orb+GGPS
TS.BTD.Pivot orb+PCG
```

where the first two are equivalent. The 3rd and 4th are more heavily on analysis and will typically not improve the bandwidth reduction.

TS.BTD.Optimize **speed|memory** *(string)*

When selecting the smallest blocks for the BTD matrix there are certain criteria that may change the size of each block. For very memory consuming jobs one may choose the **memory**.

NOTE: often both methods provide *exactly* the same BTD matrix due to constraints on the matrix.

TS.BTD.Spectral **propagation|column** *(string)*

How to compute the spectral function ($G\Gamma G^\dagger$).

For $N < 4$ this defaults to **propagation** which should be the fastest.

For $N \geq 4$ this defaults to **column**.

Check which has the best performance for your system if you endeavor on huge amounts of calculations for the same system.

TS.MUMPS.Ordering **⟨read MUMPS manual⟩** *(string)*

One may select from a number of different matrix orderings which are all described in the MUMPS manual.

The following list of orderings are available (without detailing their differences): **auto**, **AMD**, **AMF**, **SCOTCH**, **PORD**, **METIS**, **QAMD**.

TS.MUMPS.Memory **20** *(integer)*

Specify a factor for the memory consumption in MUMPS. See the **INFOG(9)** entry in the MUMPS manual. Generally if TranSIESTA dies and **INFOG(9)=-9** one should increase this number.

TS.MUMPS.BlockingFactor **112** *(integer)*

Specify the number of internal block sizes. Larger numbers increases performance at the cost of memory.

NOTE: this option may heavily influence performance.

12.6.4 Poisson solution for fixed boundary conditions

TranSIESTA requires fixed boundary conditions and forcing this is an intricate and important detail.

TS.Poisson **ramp-cell|ramp-central|elec-box|⟨file⟩** *(string)*

Define how the correction of the Poisson equation is superimposed. The default is to apply the linear correction across the entire cell (if there are two semi-infinite aligned electrodes). Otherwise this defaults to the *box* solution which will introduce spurious effects at the electrode boundaries. In this case you are encouraged to supply a **file**.

If the input is a **file**, it should be a NetCDF file containing the grid information which acts as the boundary conditions for the SCF cycle. The grid information should conform to the grid size of the unit-cell in the simulation. **NOTE:** the file option is only applicable if compiled with CDF4 compliance.

TS.Hartree.Fix **plane|elec-plane|elec-box** *(string)*

As the fixed boundary conditions requires a fixed reference potential. For two electrode calculations this defaults to taking the plane at one of the electrodes basal-planes (**plane**).

For anything but two electrodes this defaults to **elec-plane** because the plane should be at a fixed position in the cell.

NOTE: generally this shouldn't need to be changed.

TS.Hartree.Fix.Frac 1. *(real)*

Fraction of the correction that is applied.

NOTE: this is an experimental feature and shouldn't be used.

12.6.5 Electrode description options

As TranSIESTA supports N electrodes you need to specify all electrodes in a generic input format.

%block TS.Elecs ⟨**None**⟩ *(block)*

Each line denote an electrode which may be queried in **TS.Elec.<>** for its setup.

%block TS.Elec.<> ⟨**None**⟩ *(block)*

Each line represents a setting for electrode <>. There are a few lines that *must* be present, **HS**, **semi-inf-dir**, **electrode-pos**, **chem-pot**.

HS The Hamiltonian information from the initial electrode calculation. This file retains the geometrical information as well as the Hamiltonian, overlap matrix and the Fermi-level of the electrode. This is a file-path and the electrode `SystemLabel.TSHS` need not be located in the simulation folder.

semi-inf-direction|semi-inf-dir|semi-inf The semi-infinite direction of the electrode with respect to the electrode unit-cell.

NOTE: this has nothing to do with the scattering region unit cell, TranSIESTA will figure out the alignment of the electrode unit-cell and the scattering region unit-cell.

chemical-potential|chem-pot|mu The chemical potential that is associated with this elec-

trode. This is a string that should be present in the **TS.ChemPots** block.

electrode-position|elec-pos The index of the electrode in the scattering region. This may be given by either **elec-pos <idx>**, which refers to the first atomic index of the electrode residing at index **<idx>**. Else the electrode position may be given via **elec-pos end <idx>** where the last index of the electrode will be located at **<idx>**.

used-atoms Number of atoms from the electrode calculation that is used in the scattering region as electrode. This may be useful when the periodicity of the electrodes forces extensive electrodes in the semi-infinite direction.

NOTE: do not set this if you use all atoms in the electrode.

Bulk Logical controlling whether the Hamiltonian of the electrode region in the scattering region is enforced *bulk* or whether the Hamiltonian is taken from the scattering region elements.

DM-update String of values **none**, **cross-terms** or **all** which controls which part of the electrode density matrix elements that are updated. If **all**, both the density matrix elements in the electrode and the coupling elements between the electrode and scattering region are updated. If **cross-terms** only the coupling elements between the electrode and the scattering region are updated.

Gf String with filename of the surface Green function data. This may be used to place a common surface Green function file in a top directory which may then be used in all calculations using the same electrode and the same contour. If many calculations are performed this will heavily increase performance at the cost of disk-space.

Gf-Reuse Logical deciding whether the surface Green function file should be re-used or deleted. If this is **false** the surface Green function file is deleted and re-created upon start.

Eta Control the imaginary part of the surface Green function for this electrode. See **TS.Elecs.Eta**.

Accuracy Control the convergence accuracy required for the self-energy calculation when using the Lopez-Sanchez, Lopez-Sanchez iterative scheme. See **TS.Elecs.Accuracy**.

NOTE: advanced use *only*.

DE Density and energy density matrix file for the electrode. This may be used to initialize the density matrix elements in the electrode region by the bulk values. This may be used to increase the bulk-like behavior of the electrodes.

NOTE: this should only be performed on one TranSIESTA calculation as then the scattering region `SystemLabel.TSDE` contains the electrode density matrix.

Bloch 3 integers are present on this line which each denote the number of times bigger the scattering region electrode is compared to the electrode, in each lattice direction. Remark that these expansion coefficients are with regard to the electrode unit-cell. This is denoted "Bloch" because it is an expansion based on Bloch waves.

Bloch-A/a1|B/a2|C/a3 Specific Bloch expansions in each of the electrode unit-cell direction. See **Bloch** for details.

pre-expand String denoting how the expansion of the surface Green function file will be performed. This only affects the Green function file if **Bloch** is larger than 1. By default the Green function file will contain the fully expanded surface Green function, Hamiltonian and

overlap matrices (**all**). One may reduce the file size by setting this to **Green** which only expands the surface Green function. Finally **none** may be passed to reduce the file size to the bare minimum. For performance reasons **all** is preferred.

out-of-core If **true** (default) the GF files are created which contain the surface Green function. If **false** the surface Green function will be calculated when needed. Setting this to **false** will heavily degrade performance and it is highly discouraged!

check-kgrid For N electrode calculations the **k** mesh will sometimes not be equivalent for the electrodes and the device region calculations. However, TranSIESTA requires that the device and electrode **k** samplings are commensurate. This flag controls whether this check is enforced.

NOTE: only use if fully aware of the implications.

There are several flags which are globally controlling the variables for the electrodes (with **TS.Elec.<>** taking precedence).

TS.Elecs.Bulk **true** *(logical)*

This globally controls how the Hamiltonian is treated in all electrodes. See **TS.Elec.<>.Bulk**.

TS.Elecs.Eta 10^{-4} eV *(energy)*

Globally control the imaginary part used for the surface Green function calculation. See **TS.Elec.<>.Eta**.

TS.Elecs.Accuracy 10^{-13} eV *(energy)*

Globally control the accuracy required for convergence of the self-energy. See **TS.Elec.<>.Accuracy**.

TS.Elecs.Neglect.Principal **false** *(logical)*

If this is **false** TranSIESTA dies if there are connections beyond the principal cell.

NOTE: set this to **true** with care, non-physical results may arise. Use at your own risk!

TS.Elecs.Gf.Reuse **true** *(logical)*

Globally control whether the surface Green function files should be re-used (**true**) or re-created (**false**). See **TS.Elec.<>.Gf-Reuse**.

TS.Elecs.Out-of-core **true** *(logical)*

Whether the electrodes will calculate the self energy at each SCF step. Using this will not require the surface Green function files but at the cost of heavily degraded performance. You are not encouraged to set this to **false**. See **TS.Elec.<>.Out-of-core**.

TS.Elecs.DM.Update **none|cross-terms|all** *(string)*

This globally controls which parts of the electrode density matrix gets updated. See **TS.Elec.<>.DM-Update**.

TS.Elecs.DM.Init **diagon|bulk** *(string)*

The density matrix elements in the electrodes may be forcefully set to the bulk values by reading in the DM of the corresponding electrode. This may be set to **bulk** to forcefully set the bulk values.

NOTE: this should only be set to **bulk** for equilibrium calculations.

TS.Elecs.Coord.EPS 10^{-4} Bohr *(length)*

When using Bloch expansion of the self-energies one may experience difficulties in obtaining perfectly aligned electrode coordinates.

This parameter controls how strict the criteria for equivalent atomic coordinates is. If TranSIESTA crashes due to mismatch between the electrode atomic coordinates and the scattering region calculation, one may increase this criteria. This should only be done if one is sure that the atomic coordinates are almost similar and that the difference in electronic structures of the two may be negligible.

12.6.6 Chemical potentials

For N electrodes there will also be N_μ chemical potentials. They are defined via blocks similar to **TS.Elecs**.

%block TS.ChemPots ⟨**None**⟩ *(block)*

Each line denotes a new chemical potential which is defined in the **TS.ChemPot.<>** block.

%block TS.ChemPot.<> ⟨**None**⟩ *(block)*

Each line defines a setting for the chemical potential named **<>**.

chemical-shift|mu Define the chemical shift (an energy) for this chemical potential. One may specify the shift in terms of the applied bias using **V/<integer>** instead of explicitly typing the energy.

contour.eq A subblock which defines the integration curves for the equilibrium contour for this equilibrium chemical potential. One may supply as many different contours to create whatever shape of the contour

Its format is

```
contour.eq
 begin
  <contour-name-1>
  <contour-name-2>
  ...
 end
```

NOTE: If you do *not* specify **contour.eq** in the block one will automatically use the continued fraction method and you are encouraged to use 50 or more poles[**?**].

ElectronicTemperature|Temp|kT Specify the electronic temperature (as an energy or in Kelvin). This defaults to **TS.ElectronicTemperature**.

One may specify this in units of **TS.ElectronicTemperature** by using the unit **kT**.

contour.eq.pole Define the number of poles used via an energy specification. TranSIESTA will automatically convert the energy to the closest number of poles (rounding up).

NOTE: this has precedence over **TS.ChemPot.<>.contour.eq.pole.N** if it is specified *and* a positive energy. Set this to a negative energy to directly control the number of poles.

contour.eq.pole.N Define the number of poles via an integer.

NOTE: this will only take effect if **TS.ChemPot.<>.contour.eq.pole** is a negative energy.

It is important to realize that the parameterization of the voltage into the chemical potentials enables one to have a *single* input file which is never required to be changed, even when changing the applied bias.

These options complicate the input sequence for regular 2 electrode which is unfortunate.

Using `tselecs.sh -only-mu` yields this output:

```
%block TS.ChemPots
  Left
  Right
%endblock
%block TS.ChemPot.Left
  mu V/2
  contour.eq
    begin
      C-Left
      T-Left
    end
%endblock
%block TS.ChemPot.Right
  mu -V/2
  contour.eq
    begin
      C-Right
      T-Right
    end
%endblock
```

Note that the default is a 2 electrode setup with chemical potentials associated directly with the electrode names "Left"/"Right". Each chemical potential has two parts of the equilibrium contour named according to their name.

12.6.7 Complex contour integration options

Specifying the contour for N electrode systems is a bit extensive due to the possibility of more than 2 chemical potentials. Please use the `Util/TS/tselecs.sh` as a means to create default input blocks.

The contours are split in two segments. One, being the equilibrium contour of each of the different chemical potentials. The second for the non-equilibrium contour. The equilibrium contours are shifted according to their chemical potentials with respect to a reference energy. Note that for TranSIESTA the reference energy is named the Fermi-level, which is rather unfortunate (for non-equilibrium but not equilibrium). Fortunately the non-equilibrium contours are defined from different chemical potentials Fermi functions, and as such this contour is defined in the window of the minimum and maximum chemical potentials.

In this section the equilibrium contours are defined, and in the next section the non-equilibrium contours are defined.

TS.Contours.Eq.Pole 2.5 eV *(energy)*

The imaginary part of the Fermi function tail when crossing the Fermi level. Note that the actual number of poles may differ between different calculations where the electronic temperatures are different.

NOTE: if the energy specified is negative, **TS.Contours.Eq.Pole.N** takes effect.

TS.Contours.Eq.Pole.N 8 *(integer)*

Manually select the # of poles for the equilibrium contour.

NOTE: this flag will only take effect if **TS.Contours.Eq.Pole** is a negative energy.

%block TS.Contour.<> ⟨**None**⟩ *(block)*

Specify a contour named <> with options within the block.

The names <> are taken from the **TS.ChemPot.<>.contour.eq** block in the chemical potentials.

The format of this block is made up of at least 4 lines, in the following order of appearance.

part Specify which part of the equilibrium contour this is:

circle The initial circular part of the contour

square The initial square part of the contour

line The straight line of the contour

tail The final part of the contour *must* be a tail which denotes the Fermi-tail.

from *a* to *b* Define the integration range on the energy axis. Thus a and b are energies.

NOTE: that b may be supplied as **inf** for **tail** parts.

points/delta Define the number of integration points/energy separation. If specifying the number of points an integer should be supplied.

If specifying the separation between consecutive points an energy should be supplied.

method Specify the numerical method used to conduct the integration. Here a number of different numerical integration schemes are accessible

mid|mid-rule Use the mid-rule for integration.

simpson|simpson-mix Use the composite Simpson 3/8 rule (three point Newton-Cotes).

boole|boole-mix Use the composite Booles rule (five point Newton-Cotes).

G-legendre Gauss-Legendre quadrature.

NOTE: has **opt right**

tanh-sinh Tanh-Sinh quadrature.

NOTE: has **opt precision <>**.

NOTE: has **opt right**.

G-Fermi Gauss-Fermi quadrature (only on tails).

opt Specify additional options for the **method**. Only a selected subset of the methods have additional options.

These options complicate the input sequence for regular 2 electrode which is unfortunate. However, it allows highly customizable contours, etc.

Using `tselecs.sh -only-c` yields this output:

```
TS.Contours.Eq.Pole 2.5 eV
%block TS.Contour.C-Left
  part circle
   from -40. eV + V/2 to -10 kT + V/2
     points 25
      method g-legendre
%endblock
%block TS.Contour.T-Left
  part tail
   from prev to inf
     points 10
      method g-fermi
%endblock
%block TS.Contour.C-Right
  part circle
   from -40. eV -V/2 to -10 kT -V/2
     points 25
      method g-legendre
%endblock
%block TS.Contour.T-Right
  part tail
   from prev to inf
     points 10
      method g-fermi
%endblock
```

These contour options refer to input options for the chemical potentials as shown in Sec. 12.6.6 (p. 137). Importantly one should note the shift of the contours of corresponding to the chemical potential (the shift corresponds to difference from the reference energy used in TranSIESTA).

12.6.8 Bias contour integration options

The bias contour is similarly defined as the equilibrium contours. Please use the `Util/TS/tselecs.sh` as a means to create default input blocks.

TS.Contours.nEq.Eta $0\,\mathrm{eV}$ *(energy)*

The imaginary part (η) of the device states. Generally this is not necessary to define as the imaginary part arises from the self-energies (where $\eta > 0$).

TS.Contours.nEq.Fermi.Cutoff $5\,k_B T$ *(energy)*

The bias contour is limited by the Fermi function tails. Numerically it does not make sense to integrate to infinity. This energy defines where the bias integration window is turned into zero. Thus above $-|V|/2 - E$ or below $|V|/2 + E$ the DOS is defined as exactly zero.

%block TS.Contours.nEq ⟨**None**⟩ *(block)*

Each line defines a new contour on the non-equilibrium bias window. The contours defined *must* be defined in **TS.Contour.nEq.<>**.

These contours must all be **part line** or **part tail**.

%block TS.Contour.nEq.<> ⟨**None**⟩ *(block)*

This block is *exactly* equivalently defined as the **TS.Contour.<>**. See page 139.

The default options related to the non-equilibrium bias contour are defined as this:

```
%block TS.Contours.nEq
  neq-1
%endblock TS.Contours.nEq
%block TS.Contour.nEq.neq-1
  part line
   from -|V|/2 - 5 kT to |V|/2 + 5 kT
     delta 0.01 eV
      method mid-rule
%endblock TS.Contour.nEq.neq-1
```

If one chooses a different reference energy than 0, then the limits should change accordingly. Note that here **kT** refers to **TS.ElectronicTemperature**.

12.7 Matching TranSIESTA coordinates: basic rules

Having discussed the possible input options of TranSIESTA here we just list a set of rules to construct the appropriate coordinates of the scattering region. Contrary to versions pre 4.1, the order of atoms is largely irrelevant. One may define all electrodes, then subsequently the device, or vice versa. Similarly are buffer atoms not restricted to be the first/last atoms. However, each electrode atoms *must* be defined consecutively. I.e. if an electrode input option is given by:

```
%block TS.Elec.<>
  HS ../elec-<>/siesta.TSHS
  bloch 1 3 1
  used-atoms 4
  electrode-position 10
%endblock
```

then the atoms from 10 to 21 must coincide with the atoms of the calculation performed in the `../elec-<>/` subdirectory. The Bloch expansion requires a particular sequence of the atoms which may be outlined as in the following loop:

```
iaD = 10 ! as per the above input option
do iaE = 1 , na_u
  do iC = 0 , nC - 1
  do iB = 0 , nB - 1
  do iA = 0 , nA - 1
    xyz_device(:, iaD) = xyz_elec(:, iaE) + &
        cell_elec(:, 1) * iA + &
        cell_elec(:, 2) * iB + &
        cell_elec(:, 3) * iC
    iaD = iaD + 1
  end do
  end do
  end do
end do
```

As a help, TranSIESTA prints out the expected coordinates as though the first device atom coincides with the first electrode atom. Another means to create this is using the sisl program and this command line:

```
sgeom -rx 1 -ry 3 -rz 1 ELEC.fdf DEVICE_ELEC.fdf
```

and then shift the coordinates according to the placement in the device region.

12.8 Output

TranSIESTA generates several output files.

`SystemLabel.DM` : The SIESTA density matrix. SIESTA initially performs a calculation at zero bias assuming periodic boundary conditions in all directions, and no voltage, which is used as a starting point for the TranSIESTA calculation.

`SystemLabel.TSDE` : The TranSIESTA density matrix and energy density matrix. During a transiesta run, the `SystemLabel.DM` values are used for the density matrix in the buffer (if used) and electrode regions. The coupling terms may or may not be updated in a TranSIESTA run, see **TS.Elec.<>.DM-Update**.

`SystemLabel.TSHS` : The Hamiltonian corresponding to `SystemLabel.TSDE`. This file also contains geometry information etc. needed by TranSIESTA and TBtrans.

`SystemLabel.TSKP` : The k-points used in the TranSIESTA calculation. See SIESTA `SystemLabel.KP` file for formatting information.

`SystemLabel.TSCCEQ*` : The equilibrium complex contour integration paths.

`SystemLabel.TSCCNEQ*` : The non-equilibrium complex contour integration paths.

12.9 Utilities for analysis: TBtrans

Please see the separate TBtrans manual (tbtrans.pdf).

13 ANALYSIS TOOLS

There are a number of analysis tools and programs in the `Util` directory. Some of them have been directly or indirectly mentioned in this manual. Their documentation is the appropriate sub-directory of `Util`. See `Util/README`.

14 SCRIPTING

In the `Util/Scripting` directory we provide an experimental python scripting framework built on top of the "Atomic Simulation Environment" (see `https://wiki.fysik.dtu.dk/ase2`) by the

Campos group at DTU, Denmark.

(NOTE: “ASE version 2”, not the new version 3, is needed)

There are objects implementing the "Siesta as server/subroutine" feature, and also hooks for file-oriented-communication usage. This interface is different from the SIESTA-specific functionality already contained in the ASE framework.

Users can create their own scripts to customize the “outer geometry loop” in SIESTA, or to perform various repetitive calculations in compact form.

Note that the interfaces in this framework are still evolving and are subject to change.

Suggestions for improvements can be sent to Alberto Garcia (albertog@icmab.es)

15 PROBLEM HANDLING

15.1 Error and warning messages

`chkdim: ERROR: In` *`routine`* `dimension` *`parameter = value.`* `It must be ...` And other similar messages.

Description: Some array dimensions which change infrequently, and do not lead to much memory use, are fixed to oversized values. This message means that one of this parameters is too small and neads to be increased. However, if this occurs and your system is not very large, or unusual in some sense, you should suspect first of a mistake in the data file (incorrect atomic positions or cell dimensions, too large cutoff radii, etc).

Fix: Check again the data file. Look for previous warnings or suspicious values in the output. If you find nothing unusual, edit the specified routine and change the corresponding parameter.

16 REPORTING BUGS

Your assistance is essential to help improve the program. If you find any problem, or would like to offer a suggestion for improvement, please follow the instructions in the file `Docs/REPORTING_BUGS`.

Since SIESTA has moved to Launchpad you are encouraged to follow the instructions presented at: `https://answers.launchpad.net/siesta/+faq/2779`.

17 ACKNOWLEDGMENTS

We want to acknowledge the use of a small number of routines, written by other authors, in developing the siesta code. In most cases, these routines were acquired by now-forgotten routes, and the reported authorships are based on their headings. If you detect any incorrect or incomplete attribution, or suspect that other routines may be due to different authors, please let us know.

- The main nonpublic contribution, that we thank thoroughly, are modified versions of a number of routines, originally written by **A. R. Williams** around 1985, for the solution of the radial Schrödinger and Poisson equations in the APW code of Soler and Williams (PRB **42**, 9728

(1990)). Within SIESTA, they are kept in files arw.f and periodic_table.f, and they are used for the generation of the basis orbitals and the screened pseudopotentials.

- The exchange-correlation routines contained in file xc.f were written by J.M.Soler in 1996 and 1997, in collaboration with **C. Balbás** and **J. L. Martins**. Routine pzxc (in the same file), which implements the Perdew-Zunger LDA parametrization of xc, is based on routine velect, written by **S. Froyen**.
- The serial version of the multivariate fast fourier transform used to solve Poisson's equation was written by **Clive Temperton**.
- Subroutine iomd.f for writing MD history in files was originally written by **J. Kohanoff**.

We want to thank very specially **O. F. Sankey**, **D. J. Niklewski** and **D. A. Drabold** for making the FIREBALL code available to P. Ordejón. Although we no longer use the routines in that code, it was essential in the initial development of the SIESTA project, which still uses many of the algorithms developed by them.

We thank **V. Heine** for his support and encouraging us in this project.

The SIESTA project is supported by the Spanish DGES through several contracts. We also acknowledge past support by the Fundación Ramón Areces.

18 APPENDIX: Physical unit names recognized by FDF

Magnitude	Unit name	MKS value
mass	Kg	1.E0
mass	g	1.E-3
mass	amu	1.66054E-27
length	m	1.E0
length	cm	1.E-2
length	nm	1.E-9
length	Ang	1.E-10
length	Bohr	0.529177E-10
time	s	1.E0
time	fs	1.E-15
time	ps	1.E-12
time	ns	1.E-9
time	mins	60.E0
time	hours	3.6E3
time	days	8.64E4
energy	J	1.E0
energy	erg	1.E-7
energy	eV	1.60219E-19
energy	meV	1.60219E-22
energy	Ry	2.17991E-18
energy	mRy	2.17991E-21
energy	Hartree	4.35982E-18
energy	K	1.38066E-23
energy	kcal/mol	6.94780E-21
energy	mHartree	4.35982E-21
energy	kJ/mol	1.6606E-21
energy	Hz	6.6262E-34
energy	THz	6.6262E-22
energy	cm-1	1.986E-23
energy	cm**-1	1.986E-23
energy	cmˆ -1	1.986E-23
force	N	1.E0
force	eV/Ang	1.60219E-9
force	Ry/Bohr	4.11943E-8

Magnitude	Unit name	MKS value
pressure	Pa	1.E0
pressure	MPa	1.E6
pressure	GPa	1.E9
pressure	atm	1.01325E5
pressure	bar	1.E5
pressure	Kbar	1.E8
pressure	Mbar	1.E11
pressure	Ry/Bohr**3	1.47108E13
pressure	eV/Ang**3	1.60219E11
charge	C	1.E0
charge	e	1.602177E-19
dipole	C*m	1.E0
dipole	D	3.33564E-30
dipole	debye	3.33564E-30
dipole	e*Bohr	8.47835E-30
dipole	e*Ang	1.602177E-29
MomInert	Kg*m**2	1.E0
MomInert	Ry*fs**2	2.17991E-48
Efield	V/m	1.E0
Efield	V/nm	1.E9
Efield	V/Ang	1.E10
Efield	V/Bohr	1.8897268E10
Efield	Ry/Bohr/e	2.5711273E11
Efield	Har/Bohr/e	5.1422546E11
angle	deg	1.d0
angle	rad	5.72957795E1
torque	eV/deg	1.E0
torque	eV/rad	1.745533E-2
torque	Ry/deg	13.6058E0
torque	Ry/rad	0.237466E0
torque	meV/deg	1.E-3
torque	meV/rad	1.745533E-5
torque	mRy/deg	13.6058E-3
torque	mRy/rad	0.237466E-3

19 APPENDIX: XML Output

From version 2.0, SIESTA includes an option to write its output to an XML file. The XML it produces is in accordance with the CMLComp subset of version 2.2 of the Chemical Markup Language. Further information and resources can be found at `http://cmlcomp.org/` and tools for working with the XML file can be found in the `Util/CMLComp` directory.

The main motivation for standarised XML (CML) output is as a step towards standarising formats for uses like the following.

- To have SIESTA communicating with other software, either for postprocessing or as part of a larger workflow scheme. In such a scenario, the XML output of one SIESTA simulation may be easily parsed in order to direct further simulations. Detailed discussion of this is out of the scope of this manual.
- To generate webpages showing SIESTA output in a more accessible, graphically rich, fashion. This section will explain how to do this.

19.1 Controlling XML output

XML.Write **true** *(logical)*

Determine if the main XML file should be created for this run.

19.2 Converting XML to XHTML

The translation of the SIESTA XML output to a HTML-based webpage is done using XSLT technology. The stylesheets conform to XSLT-1.0 plus EXSLT extensions; an xslt processor capable of dealing with this is necessary. However, in order to make the system easy to use, a script called ccViz is provided in `Util/CMLComp` that works on most Unix or Mac OS X systems. It is run like so:

```
./ccViz SystemLabel.xml
```

A new file will be produced. Point your web-browser at `SystemLabel.xhtml` to view the output.

The generated webpages include support for viewing three-dimensional interactive images of the system. If you want to do this, you will either need jMol (`http://jmol.sourceforge.net`) installed or access to the internet. As this is a Java applet, you will also need a working Java Runtime Environment and browser plugin - installation instructions for these are outside the scope of this manual, though. However, the webpages are still useful and may be viewed without this plugin.

An online version of this tool is avalable from `http://cmlcomp.org/ccViz/`, as are updated versions of the ccViz script.

20 APPENDIX: Selection of precision for storage

Some of the real arrays used in Siesta are by default single-precision, to save memory. This applies to the array that holds the values of the basis orbitals on the real-space grid, to the historical data sets in Broyden mixing, and to the arrays used in the O(N) routines. Note that the grid functions (charge densities, potentials, etc) are now (since mid January 2010) in double precision by default.

The following pre-processing symbols at compile time control the precision selection

- Add `-DGRID_SP` to the `DEFS` variable in `arch.make` to use single-precision for all the grid magnitudes, including the orbitals array and charge densities and potentials. This will cause some numerical differences and will have a negligible effect on memory consumption, since the orbitals array is the main user of memory on the grid, and it is single-precision by default. This setting will recover the default behavior of versions prior to 4.0.
- Add `-DGRID_DP` to the `DEFS` variable in `arch.make` to use double-precision for all the grid magnitudes, including the orbitals array. This will significantly increase the memory used for large problems, with negligible differences in accuracy.
- Add `-DBROYDEN_DP` to the `DEFS` variable in `arch.make` to use double-precision arrays for the Broyden historical data sets. (Remember that the Broyden mixing for SCF convergence acceleration is an experimental feature.)
- Add `-DON_DP` to the `DEFS` variable in `arch.make` to use double-precision for all the arrays in the O(N) routines.

21 APPENDIX: Data structures and reference counting

To implement some of the new features (e.g. charge mixing and DM extrapolation), SIESTA uses new flexible data structures. These are defined and handled through a combination and extension of ideas already in the Fortran community:

- Simple templating using the "include file" mechanism, as for example in the FLIBS project led by Arjen Markus (`http://flibs.sourceforge.net`).
- The classic reference-counting mechanism to avoid memory leaks, as implemented in the PyF95++ project (`http://blockit.sourceforge.net`).

Reference counting makes it much simpler to store data in container objects. For example, a circular stack is used in the charge-mixing module. A number of future enhancements depend on this paradigm.